Faster Algorithms for Longest Common Substring

PANAGIOTIS CHARALAMPOPOULOS, King's College London, London, UK
TOMASZ KOCIUMAKA, Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
JAKUB RADOSZEWSKI, Institute of Informatics, University of Warsaw, Warsaw, Poland and Samsung R&D Institute Poland, Warsaw, Poland
SOLON P. PISSIS, CWI, Amsterdam, The Netherlands and Vrije Universiteit, Amsterdam, The Netherlands

In the classic Longest Common Substring (LCS) problem, we are given two strings S and T of total length n over an alphabet of size σ, and we are asked to find a longest string occurring as a fragment of both S and T. Weiner, in his seminal paper that introduced the suffix tree, presented an $O(n \log \sigma)$-time algorithm for the LCS problem [SWAT 1973]. For polynomially-bounded integer alphabets, the linear-time construction of suffix trees by Farach yielded an $O(n)$-time algorithm for the LCS problem [FOCS 1997]. However, for small alphabets, this is not necessarily optimal for the LCS problem in the word RAM model of computation, in which the strings can be stored in $O(n \log \sigma / \log n)$ space and read in $O(n \log \sigma / \log n)$ time. We show that we can compute an LCS of two strings in time $O(n \log \sigma / \sqrt{\log n})$ in the word RAM model, which is sublinear in n if $\sigma = 2^{o(\sqrt{\log n})}$ (in particular, if $\sigma = O(1)$), using optimal space $O(n \log \sigma / \log n)$. In fact, it was recently shown that this result is conditionally optimal [Kempa and Kociumaka, STOC 2025]. The same complexity can be achieved for computing an LCS of $\lambda = O(\sqrt{\log n}/\log\log n)$ input strings of total length n.

We then lift our ideas to the problem of computing a k-mismatch LCS, which has received considerable attention in recent years. In this problem, the aim is to compute a longest substring of S that occurs in T with at most k mismatches. Flouri et al. showed how to compute a 1-mismatch LCS in $O(n \log n)$ time [IPL 2015]. Thankachan et al. extended this result to computing a k-mismatch LCS in $O(n \log^k n)$ time for $k = O(1)$ [J. Comput. Biol. 2016]. We show an $O(n \log^{k-0.5} n)$-time algorithm, for any constant integer $k > 0$ and *irrespective* of the alphabet size, using $O(n)$ space as the previous approaches. We thus notably break through the well-known $n \log^k n$ barrier, which stems from a recursive heavy-path decomposition technique that was first introduced in the seminal paper of Cole et al. for string indexing with k errors [STOC 2004].

This is an extended version of an article that was published in the proceedings of the 29th Annual European Symposium on Algorithms (ESA 2021).

P. Charalampopoulos was partly supported by the Israel Science Foundation grant 592/17. T. Kociumaka was partly supported by NSF 1652303, 1909046 and HDR TRIPODS 1934846 grants and an Alfred P. Sloan Fellowship. J. Radoszewski was supported by the Polish National Science Center, grant nos. 2018/31/D/ST6/03991 and 2022/46/E/ST6/00463. S. P. Pissis was supported in part by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements nos. 872539 and 956229, respectively.

Authors' Contact Information: Panagiotis Charalampopoulos, King's College London, London, UK; e-mail: p.charalampopoulos@kcl.ac.uk; Tomasz Kociumaka, Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany; e-mail: tomasz.kociumaka@mpi-inf.mpg.de; Jakub Radoszewski, Institute of Informatics, University of Warsaw, Warsaw, Poland and Samsung R&D Institute Poland, Warsaw, Poland; e-mail: jrad@mimuw.edu.pl; Solon P. Pissis (corresponding author), CWI, Amsterdam, The Netherlands and Vrije Universiteit, Amsterdam, The Netherlands; e-mail: solon.pissis@cwi.nl.

ACM 1549-6333/2026/2-ART19
https://doi.org/10.1145/3774754

As a by-product, we improve upon the algorithm of Charalampopoulos et al. [CPM 2018] for computing a k-mismatch LCS in the case when the output k-mismatch LCS is sufficiently long.

CCS Concepts: • **Theory of computation → Pattern matching**;

Additional Key Words and Phrases: longest common substring, k mismatches, wavelet tree

ACM Reference format:
Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, and Solon P. Pissis. 2026. Faster Algorithms for Longest Common Substring. *ACM Trans. Algor.* 22, 2, Article 19 (February 2026), 47 pages.
https://doi.org/10.1145/3774754

1 Introduction

In the classic **longest common substring (LCS)** problem, we are given two strings S and T of total length n over an alphabet of size σ, and we are asked to compute a longest string occurring as a fragment of both S and T (see Figure 1 for an example). The problem was conjectured by Knuth to require $\Omega(n \log n)$ time until Weiner, in his seminal paper introducing the suffix tree [93], showed that the LCS problem can be solved in $O(n)$ time when σ is constant via constructing the suffix tree of string $S\#T$ for a sentinel letter #. Later, Farach showed that even if σ is not constant, the suffix tree can be constructed in linear time in addition to the time required for sorting the characters of S and T [52]. This yielded an $O(n)$-time algorithm for the LCS problem in the word RAM model for polynomially-bounded integer alphabets. While Farach's algorithm for suffix tree construction is *optimal* for all alphabets (the suffix tree by definition has size $\Theta(n)$), optimality does not carry over to the LCS problem. We were thus motivated to answer the following basic question:

$$\textit{Can the LCS problem be solved in } o(n) \textit{ time when } \log\sigma = o(\log n)?$$

We consider the word RAM model and assume an alphabet $[0\,..\,\sigma) = \{0, 1, \ldots, \sigma-1\}$. Any string of length n can then be stored in $O(n \log\sigma/\log n)$ space and read in $O(n \log\sigma/\log n)$ time. We answer the above question positively when $\log\sigma = o(\sqrt{\log n})$:

Theorem 1.1. *Given two strings S and T of total length n over an alphabet $[0\,..\,\sigma)$, the LCS problem can be solved in $O(n \log\sigma/\sqrt{\log n})$ time using $O(n \log\sigma/\log n)$ space.*

In a work subsequent to ours, Kempa and Kociumaka [70] showed that a variant of the dictionary matching problem underlies the hardness of a plethora of problems in the word RAM model. In particular, they showed that the time complexity in Theorem 1.1 is optimal conditioned to said problem for all non-unary alphabets. Notably, as a by-product of their hardness reduction, they obtain a reduction from any instance of the LCS problem with strings of total length $O(n)$ under alphabet $[0\,..\,\sigma)$ to an instance of the LCS problem with strings of total length $O(n \log\sigma)$ under the binary alphabet.

The classic solution for the LCS problem over integer alphabets allows to compute an LCS of $O(1)$ strings of total length n in $O(n)$ time; with extra care, it can also be extended to arbitrarily many strings in the same time complexity [64]. We show that the result of Theorem 1.1 can be achieved also for computing an LCS of $\lambda = O(\sqrt{\log n}/\log\log n)$ input strings of total length n (cf. Theorem 5.8).

We also consider the following generalisation of the LCS problem that allows for mismatches (see Figure 1 for an example).

Fig. 1. Left: The LCS of the two strings has length 5. Right: The 1-LCS (1-mismatch LCS) of the same two strings has length 7; the mismatching letters are shown in red.

k-Mismatch Longest Common Substring (k-LCS)
Input: Two strings S and T of total length n and an integer $k > 0$.
Output: A pair S', T' of substrings of S and T, respectively, with Hamming distance (i.e., number of mismatches) at most k and maximal length $|S'| = |T'|$.

Flouri et al. presented an $O(n \log n)$-time algorithm for the 1-LCS problem [54]. (Earlier work on this problem includes [10].) This was generalised by Thankachan et al. [89] to an algorithm for the k-LCS problem that works in $O(n \log^k n)$ time if $k = O(1)$. Both algorithms use $O(n)$ space. In [34], Charalampopoulos et al. presented an $O(n+n \log^{k+1} n/\sqrt{\ell})$-time algorithm for k-LCS with $k = O(1)$, where ℓ is the length of a k-LCS. For general k, Flouri et al. presented an $O(n^2)$-time algorithm that uses $O(1)$ additional space [54]. Grabowski [61] presented two algorithms with running times $O(n((k+1)(\ell_0+1))^k)$ and $O(n^2k/\ell_k)$, where ℓ_0 and ℓ_k are, respectively, the length of an LCS of S and T and the length of a k-LCS of S and T. Abboud et al. [1] employed the polynomial method to obtain a $k^{1.5}n^2/2^{\Omega(\sqrt{\log n/k})}$-time randomised algorithm. In [72], Kociumaka et al. showed that, assuming the **Strong Exponential Time Hypothesis (SETH)** [66, 67], no strongly subquadratic-time solution for k-LCS exists for $k = \Omega(\log n)$. The authors of [72] additionally presented an $O(n^{1.5} \log^2 n)$-time 2-approximation algorithm for k-LCS for general k; it was improved in [60] to run in $O(n^{4/3+o(1)})$ time for strings over a constant-sized alphabet.

Analogously to Weiner's solution to the LCS problem via suffix trees, the algorithm of Thankachan et al. [89] builds upon the ideas of the k-errata tree, which was introduced by Cole et al. [46] in their seminal paper for indexing a string of length n with the aim of answering pattern matching queries with up to k mismatches (that is, under the Hamming distance) or errors (that is, under the edit distance). For constant k, the size of the k-errata tree is $O(n \log^k n)$. Let us stress that computing a k-LCS using the k-errata tree directly is not straightforward, as opposed to computing an LCS using the suffix tree.

We show the following result, breaking through the $n \log^k n$ barrier for all constant integers $k > 0$ irrespectively of the alphabet size. Recall that, in the word RAM model, the letters of S and T can be renumbered in $O(n \log \log n)$ time [63] so that they belong to $[0\,..\,\sigma)$, where σ is the total number of distinct letters in S and T.

Theorem 1.2. *Given two strings S and T of total length n and a constant integer $k > 0$, the k-LCS problem can be solved in $O(n \log^{k-1/2} n)$ time using $O(n)$ space.*

Notably, on the way to proving the above theorem, we improve upon [34] by showing an $O(n + \frac{n}{\ell} \log^{k+1} n)$-time algorithm for k-LCS with $k = O(1)$, where ℓ is the length of a k-LCS; see Corollary 6.7. (Our second summand is smaller by a $\sqrt{\ell}$ multiplicative factor compared to [34].)

1.1 Our Techniques

At the heart of our approaches lies the following Two String Families LCP Problem. Here, the length of the longest common prefix of two strings U and V is denoted by $\mathsf{LCP}(U, V)$; for a precise definition of compacted tries, see Section 2.

Two String Families LCP Problem
Input: A compacted trie $\mathcal{T}(\mathcal{F})$ of $\mathcal{F} \subseteq \Sigma^*$ and two sets $\mathcal{P}, \mathcal{Q} \subseteq \mathcal{F}^2$ whose elements are represented by pairs of pointers to nodes of $\mathcal{T}(\mathcal{F})$, with $|\mathcal{P}|, |\mathcal{Q}|, |\mathcal{F}| \leq N$.
Output: $\text{maxPairLCP}(\mathcal{P}, \mathcal{Q}) = \max\{\mathsf{LCP}(P_1, Q_1) + \mathsf{LCP}(P_2, Q_2) : (P_1, P_2) \in \mathcal{P}, (Q_1, Q_2) \in \mathcal{Q}\}$.

This abstract problem was introduced in [34]. The solution encapsulated in the following lemma is directly based on a technique that was used in [27, 49] and then in [54] to devise an $O(n \log n)$-time solution for 1-LCS.

Lemma 1.3 ([34, Lemma 3]). *The Two String Families LCP Problem can be solved in $O(N \log N)$ time and $O(N)$ space*[1].

In particular, Lemma 1.3 implies an $O(n \log n)$-time algorithm for 1-LCS; see the following example.

Example 1.4. For a string T, let $T[i \,..\, j]$ denote the fragment of T that starts at position i (1-based) and ends at position j, and let T^R be T reversed. For two strings S and T, we define the following sets of string pairs:

$$\mathcal{P} = \{((S[1 \,..\, i-1])^R, S[i+1 \,..\, |S|]) \,:\, i \in [1 \,..\, |S|]\}$$
$$\mathcal{Q} = \{((T[1 \,..\, i-1])^R, T[i+1 \,..\, |T|]) \,:\, i \in [1 \,..\, |T|]\}.$$

Either the 1-LCS of S and T is simply their LCS, or the length of the 1-LCS of S and T equals $\text{maxPairLCP}(\mathcal{P}, \mathcal{Q})$. The underlying set $\mathcal{F}$ in the instance of Two String Families LCP Problem is the set of suffixes of S, T, S^R and T^R. The compacted trie $\mathcal{T}(\mathcal{F})$ can be inferred in $O(n)$ time from the suffix tree of $S\#_1 T\#_2 S^R\#_3 T^R$ for sentinel letters $\#_1, \#_2, \#_3$. Therefore, indeed Lemma 1.3 implies an $O(n \log n)$-time algorithm for 1-LCS.

In the algorithm underlying Lemma 1.3, for each node v of $\mathcal{T}(\mathcal{F})$, we try to identify a pair of elements, one from $\mathcal{P}$ and one from $\mathcal{Q}$, whose first components are descendants of v and the LCP of their second components is maximised. The algorithm traverses $\mathcal{T}(\mathcal{F})$ bottom-up and uses mergeable height-balanced trees with $O(N \log N)$ total merging time to store elements of pairs; see [28].

No $o(N \log N)$ time solution to the Two String Families LCP Problem is known, and devising such an algorithm seems difficult. The key ingredient of our algorithms is an efficient solution to a special case of the problem where we have bounds on the lengths of the involved strings. We say that a family $\mathcal{X}$ of string pairs is an (α, β)-*family* if each $(U, V) \in \mathcal{X}$ satisfies $|U| \leq \alpha$ and $|V| \leq \beta$.

Lemma 1.5. *An instance of the Two String Families LCP Problem in which $\mathcal{P}$ and $\mathcal{Q}$ are (α, β)-families can be solved in time $O(N(\alpha + \log N)(\log \beta + \sqrt{\log N})/\log N)$ and space $O(N + N\alpha/\log N)$.*

Observe that the algorithm of Lemma 1.5 works in $O(N\sqrt{\log N})$ time if $\alpha = O(\log N)$ and $\log \beta = O(\sqrt{\log N})$. The algorithm uses a wavelet tree (cf. [62]) of the first components of $\mathcal{P} \cup \mathcal{Q}$.

Solution to LCS. For the LCS problem, we design three different algorithms and pick one depending on the length of the solution. For short LCS ($\leq \frac{1}{3} \log_\sigma n$), we employ a simple tabulation technique. For long LCS ($\geq \log^4 n$), we obtain a small instance of the Two String Families LCP Problem by

[1]The original formulation of [34, Lemma 3] does not include a claim about the space complexity. However, it can be readily verified that the underlying algorithm, described in [49, 54], uses only linear space.

employing difference covers [45, 77], similarly to the solution of Charalampopoulos et al. [34] for the long k-LCS problem. The main modification in this part is the usage of the sublinear **Longest Common Extension (LCE)** data structure of Kempa and Kociumaka [69]. Our solutions for short LCS and long LCS work in $\mathcal{O}(n/\log_\sigma n)$ time.

As for medium-length LCS ($\geq \frac{1}{3}\log_\sigma n$ and $\leq \beta = 2^{\mathcal{O}(\sqrt{\log n})}$), let us first consider a case when the strings do not contain highly periodic fragments. In this case, we use the string synchronising sets of Kempa and Kociumaka [69] to select a set of $\mathcal{O}(\frac{n}{\tau})$ anchors over S and T, where $\tau = \Theta(\log_\sigma n)$, such that, for any common substring U of S and T of length $\ell \geq 3\tau - 1$, there exist occurrences $S[i^S \,..\, j^S]$ and $T[i^T \,..\, j^T]$ of U, for which we have anchors $a^S \in [i^S \,..\, j^S]$ and $a^T \in [i^T \,..\, j^T]$ with $a^S - i^S = a^T - i^T \leq \tau$. For each anchor a in S, we add a string pair $((S[a-\tau \,..\, a))^R, S[a \,..\, a+\beta))$, possibly truncated at the ends of the string, to $\mathcal{P}$ (and similarly for T and $\mathcal{Q}$). This lets us apply Lemma 1.5 with $N = \mathcal{O}(n/\tau)$, $\alpha = \mathcal{O}(\tau)$ and $\beta = 2^{\mathcal{O}(\sqrt{\log n})}$. In the periodic case, we cannot guarantee that $a^S - i^S = a^T - i^T$ is small, but we can obtain a different set of anchors based on runs (maximal repetitions; cf. [74]) that yields multiple instances of the Two String Families LCP Problem, which have extra structure leading to a linear-time solution.

Generalisation to Many Strings. We allow $\lambda = \mathcal{O}(\sqrt{\log n}/\log\log n)$ input strings. Here, we also consider three cases based on the LCS length. The solution to short LCS is basically the same. For long LCS, we generalise difference covers from two to many strings, which might be a result of independent interest. More precisely, we say that (D, h) is a (λ, d)-*cover* if $D \subseteq \mathbb{Z}_+$ is a set and $h : \mathbb{Z}_+^\lambda \to [0 \,..\, d)$ is an efficiently computable function such that, for any $i_1, \ldots, i_\lambda \in \mathbb{Z}_+$ and $t \in [1 \,..\, \lambda]$, we have $i_t + h(i_1, \ldots, i_\lambda) \in D$. The construction of (λ, d)-covers generalises the original construction behind d-covers [45, 77].

Theorem 1.6. *For every positive integers $\lambda \geq 2$ and d, there is a (λ, d)-cover (D, h) such that the set $D \cap [1 \,..\, n]$ is of size $\mathcal{O}(\lambda \cdot n/\sqrt[\lambda]{d})$. After $\mathcal{O}(\log d)$-time initialisation, the set $D \cap [1 \,..\, n]$ can be constructed in $\mathcal{O}(\lambda \cdot n/\sqrt[\lambda]{d})$ time and the function $h(i_1, \ldots, h_\lambda)$ can be evaluated in $\mathcal{O}(\lambda)$ time.*

Further, we generalise the Two String Families LCP Problem to λ string families and obtain a counterpart of Lemma 1.3 for this problem with a simpler but slower $\mathcal{O}(n \log^{\mathcal{O}(1)} n)$-time solution; this is still sufficient if the threshold for a long LCS is adjusted properly. We achieve this by opening the black box behind Lemma 1.3 and extending a greedy property that stands behind it from two to λ input strings.

For medium-length LCS, the application of string synchronising sets and runs does not require any essential changes. Lemma 1.5 can be carefully extended to λ strings using said greedy property. Our solution to the special instances of the Two String Families LCP Problem that arise due to periodicity in the $\lambda = \mathcal{O}(\sqrt{\log n}/\log\log n)$ input strings requires dynamic predecessor data structures. We keep the space $\mathcal{O}(n/\log_\sigma n)$ (and the solution deterministic) by applying a space-efficient version of van Emde Boas trees [92] described in [75, 95].

Solution to k-LCS. In this case, we also obtain a set of $\mathcal{O}(n/\ell)$ anchors, where ℓ is the length of a k-LCS. If the common substring is far from highly periodic, we use a string synchronising set for $\tau = \Theta(\ell)$, and otherwise we generate anchors using a technique of misperiods that was initially introduced for k-mismatch pattern matching [26, 40].

Now, the families $\mathcal{P}, \mathcal{Q}$ need to consist not simply of substrings of S and T but rather of modified substrings generated by an approach that resembles k-errata trees [46]. This requires combining the ideas of Thankachan et al. [89]—who showed how to transform a family of strings into a family of modified strings such that LCP_k values for the former correspond to (maximal) LCP values for the latter—and (indirectly) Charalampopoulos et al. [34]—who lifted this idea to families consisting

of pairs of strings that stem from 'sitting on anchors'. Limiting the space usage of the algorithm to $O(n)$ proved to be quite technically challenging; note that, for example, this was not an issue in [34] where only a long enough LCS was sought. Finally, we apply Lemma 1.3 or Lemma 1.5 depending on the length ℓ, breaking through the $n \log^k n$ barrier for k-LCS.

1.2 Related Work

1.2.1 Longest Common Substring. Other results on the LCS problem include trade-offs between the time and the working space for computing an LCS of two strings [20, 73, 86], internal LCS queries [6], computing the LCS of two compressed strings [58, 79], the dynamic maintenance of an LCS [4–6, 36] and the so-called heaviest induced ancestors queries that are a common theme of compressed and dynamic LCS [2, 35]. The (long) LCS problem has also been recently considered in the quantum setting [76], with almost optimal algorithms presented in [3, 68].

A version of the k-LCS problem in which it suffices to compute an LCS with *approximately* k mismatches has also been considered [60, 72]. The k-LCS problem has also been studied under edit distance. In this case, an $O(n \log^k n)$-time algorithm is known [88] (cf. [8] for an efficient average-case algorithm). It remains open if our approach can be used to improve upon the algorithm from [88]. This would require substantial changes in the algorithm; a promising approach would be to use locked fragments (see [41, 47]) instead of misperiods for computing anchors (see Lemma 6.2).

1.2.2 Stringology in the Word RAM Model. A large body of work has been devoted to exploiting bit parallelism in the word RAM model for pattern matching [11, 15, 19, 22, 23, 25, 30, 55, 56, 59, 71, 85, 87]. The problem of indexing a string of length n over an alphabet $[0\,..\,\sigma)$ in the word RAM model, with the aim of efficiently answering pattern matching queries, has attracted significant attention [14, 17, 18, 24, 69, 80–82, 84]. Since, by definition, the suffix tree occupies $\Theta(n)$ space, alternative indexes have been sought. The state of the art is an index that occupies $O(n \log \sigma / \log n)$ space and can be constructed in $O(n \log \sigma / \sqrt{\log n})$ time; see Kempa and Kociumaka [69] and Munro et al. [82]. Interestingly, the running time of our algorithm (Theorem 1.1) matches the construction time of this index. Note that, in contrast to suffix trees, such indexes *cannot be used directly* for computing an LCS. Intuitively, these indexes sample suffixes of the string to be indexed and upon a pattern matching query, they treat separately the first $O(\log_\sigma n)$ letters of the pattern.

Bit parallelism was also considered in the context of other problems in stringology [12, 42, 50].

1.2.3 k-Mismatch Stringology Problems. There are other problems in computational biology that allow k mismatches and for which $O(n \log^k n)$-time algorithms are known. This includes the k-mismatch average common substring [89] and k-mappability [37]. It remains open if our techniques can be applied to these problems as well. Intuitively, the difficulty lies in that these problems seem to require computing maximal approximate LCP values for every suffix of the text separately.

As for k-mismatch indexing, for $k = O(1)$, a k-errata tree occupies $O(n \log^k n)$ space, can be constructed in $O(n \log^k n)$ time, and supports pattern matching queries with at most k mismatches in $O(m + \log^k n \log \log n + occ)$ time, where m is the length of the pattern and occ is the number of the reported pattern occurrences. Other trade-offs for this problem, in which the product of space and query time is $\Omega(n \log^{2k} n)$, were shown in [32, 90] (a product $\Omega(n \log^{2k-1} n)$ of space and query time can be achieved for strings over constant-sized alphabets [31]), and solutions with $O(n)$ space but $\Omega(\min\{n, \sigma^k m^{k-1}\})$-time queries were presented in [31, 43, 65, 91]. More efficient solutions for $k = 1$ are known (see [16] and references therein). Cohen-Addad et al. [44] showed that, under SETH, for $k = \Theta(\log n)$ any indexing data structure that can be constructed in polynomial time cannot have $O(n^{1-\delta})$ query time for any $\delta > 0$. They also showed that in the pointer machine

model, for the reporting version of the problem with $k = o(\log n)$, exponential dependency on k cannot be avoided in both the space and the query time. We hope that our techniques can fuel further progress in k-mismatch indexing.

1.3 Conclusions

We present $O(n \log \sigma / \sqrt{\log n})$-time and $O(n \log^{k-0.5} n)$-time (assuming $k = O(1)$) algorithms for computing the LCS and the k-LCS of two strings of total length n over an alphabet of size σ, respectively. The bottleneck of both algorithms is the construction of a wavelet tree [9, 83]. Improving the wavelet tree construction time could be a hard task; in particular, it would also improve upon the $O(n\sqrt{\log n})$-time algorithm by Chan and Pătraşcu [33] for counting inversions of a permutation; cf. [69]. As already mentioned, there is a matching conditional lower bound for LCS; see [70]. However, in the case of the k-LCS problem, we are not aware of such a lower bound.

1.4 Preliminary Version

A preliminary version of this work was presented in [39]. The current full version extends the sublinear-time LCS computation to many input strings, has (significantly) improved exposition, and includes all the proofs.

2 Preliminaries

Strings. Let $T = T[1]T[2]\cdots T[n]$ be a *string* (or *text*) of length $n = |T|$ over an alphabet $\Sigma = [0\,..\,\sigma)$. The elements of Σ are called *letters*. By Σ^* and Σ^n we denote the sets of all finite strings and of all length-n strings, respectively.

By ε we denote the *empty string*. For two positions i and j of T, we denote by $T[i\,..\,j]$ the *fragment* of T that starts at position i and ends at position j (the fragment is empty if $i > j$). A fragment of T is represented using $O(1)$ space by specifying the indices i and j. We define $T[i\,..\,j) = T[i\,..\,j-1]$ and $T(i\,..\,j] = T[i+1\,..\,j]$. The fragment $T[i\,..\,j]$ is an *occurrence* of the underlying *substring* $P = T[i]\cdots T[j]$. We say that P occurs at *position* i in T. A *prefix* of T is a fragment of T of the form $T[1\,..\,j]$ and a *suffix* of T is a fragment of T of the form $T[i\,..\,n]$. For strings $U_1, U_2, \ldots, U_k$, by $\mathsf{LCP}(U_1, U_2, \ldots, U_k)$ we denote the length of the longest common prefix of $U_1, U_2, \ldots, U_k$. We denote the *reverse string* of T by T^R, i.e., $T^R = T[n]T[n-1]\cdots T[1]$. By UV we denote the *concatenation* of two strings U and V, i.e., $UV = U[1]U[2]\cdots U[|U|]V[1]V[2]\cdots V[|V|]$.

The lexicographic order on strings is denoted by $\preceq$. We use the following simple fact that has extensive applications, e.g., in computations using suffix arrays [78]. Fact 2.1. *For strings* $U_1 \preceq U_2 \preceq U_3$, $\mathsf{LCP}(U_1, U_3) = \min(\mathsf{LCP}(U_1, U_2), \mathsf{LCP}(U_2, U_3))$.

A positive integer p is called a *period* of a string T if $T[i] = T[i+p]$ for all $i \in [1\,..\,|T|-p]$. We refer to the smallest period as *the period* of the string, and denote it by $\mathrm{per}(T)$. A string T is called *periodic* if $\mathrm{per}(T) \le |T|/2$ and *aperiodic* otherwise. A *run* in T is a periodic fragment that cannot be extended (to the left nor to the right) without an increase of its smallest period. All runs in a string can be computed in linear time [13, 74], even over arbitrary ordered alphabets [51].

Lemma 2.2 (Periodicity Lemma (Weak Version) [53]). *If a string S has periods p and q such that $p + q \le |S|$, then $\gcd(p, q)$ is also a period of S.*

Tries. Let $\mathcal{M}$ be a finite set containing $m > 0$ strings over Σ. The *trie* of $\mathcal{M}$, denoted by $\mathcal{R}(\mathcal{M})$, contains a node for every distinct prefix of a string in $\mathcal{M}$; the root node is ε; the set of terminal nodes is $\mathcal{M}$; and edges are of the form $(u, \alpha, u\alpha)$, where u and $u\alpha$ are nodes and $\alpha \in \Sigma$. The *compacted trie* of $\mathcal{M}$, denoted by $\mathcal{T}(\mathcal{M})$, contains the root, the branching nodes and the terminal nodes of $\mathcal{R}(\mathcal{M})$. Each maximal branchless path segment from $\mathcal{R}(\mathcal{M})$ is replaced by a single edge, and a fragment of

a string $M \in \mathcal{M}$ is used to represent the label of this edge in $O(1)$ space. The best-known example of a compacted trie is the suffix tree [93]. Throughout our algorithms, $\mathcal{M}$ always consists of a set of substrings or modified substrings with $k = O(1)$ modifications (see Section 6 for a definition) of a reference string. The value $\mathsf{val}(u)$ of a node u is the concatenation of labels of edges on the path from the root to u, and the *string-depth* of u is the length of $\mathsf{val}(u)$. The size of $\mathcal{T}(\mathcal{M})$ is $O(m)$. We use the following well-known construction (cf. [48]).

Lemma 2.3. *Given a sorted list of N strings and the longest common prefixes between pairs of consecutive strings in the list, the compacted trie of the strings can be constructed in $O(N)$ time.*

Packed Strings. We assume the unit-cost word RAM model with word size $w = \Theta(\log n)$ and a standard instruction set including arithmetic operations, bitwise Boolean operations and shifts. We count the space complexity of our algorithms in machine words. The *packed representation* of a string $T \in [0\,..\,\sigma)^n$ is a list obtained by storing $\Theta(\log_\sigma n)$ letters per machine word thus representing T in $O(n/\log_\sigma n) = O(n \log \sigma / \log n)$ machine words. If T is given in the packed representation, we simply say that T is a *packed string*.

Computations on Packed Data. We use the following abstract proposition to conveniently perform sequential computations on data given in a packed form. We consider any algorithm that interacts via $O(1)$ input and output streams. In the proof, for every possible starting state of the algorithm's memory, we memoise the result of the algorithm after any τ single-bit read or write instructions. A full proof is given in Appendix A.

Proposition 2.4. *We consider a deterministic streaming algorithm A that reads $O(1)$ input streams, writes $O(1)$ output streams, uses s bits of working space and executes at most t instructions (each in $O(1)$ time) between subsequent read/write instructions. The streams are represented in a packed form with word size w. For every integer $\tau \in [1\,..\,w]$, if $s \le w$, after an $O(2^{\tau+s}(\tau t + s))$-time preprocessing, any execution of A can be simulated in $O(1 + L/\tau)$ time, where L is the total length (in bits) of all input and output streams.*

For example, we will be using the proposition as follows. If for some n the algorithm from the proposition uses $s = o(\log n)$ bits of space and executes $t = O(1)$ instructions between subsequent reads and writes, then after $o(n)$ preprocessing, it can be simulated in $O(1 + L/\log n)$ time (for $\tau = \left\lfloor \frac{1}{2} \log n \right\rfloor$).

String Synchronising Sets. Our solution uses the string synchronising sets introduced by Kempa and Kociumaka [69]. Informally, in the simpler case that T is cube-free, a τ-synchronising set of T is a small set of *synchronising positions* in T such that each length-τ fragment of T contains at least one synchronising position, and the leftmost synchronising positions within two sufficiently long matching fragments of T are consistent.

Formally, for a length-n string T and a positive integer $\tau \le \frac{1}{2}n$, a set $A \subseteq [1\,..\,n - 2\tau + 1]$ is a *τ-synchronising set* of T if it satisfies the following two conditions (see Figure 2):

(1) Consistency: If $T[i\,..\,i + 2\tau)$ matches $T[j\,..\,j + 2\tau)$, then $i \in A$ if and only if $j \in A$.
(2) Density: For $i \in [1\,..\,n - 3\tau + 2]$, $A \cap [i\,..\,i + \tau) = \emptyset$ if and only if $\mathrm{per}(T[i\,..\,i + 3\tau - 2]) \le \frac{1}{3}\tau$.

Theorem 2.5 ([69, Proposition 8.10, Theorem 8.11]). *For a string $T \in [0\,..\,\sigma)^n$ with $\sigma = n^{O(1)}$ and $\tau \le \frac{1}{2}n$, there exists a τ-synchronising set of size $O(n/\tau)$ that can be constructed in $O(n)$ time or, if $\tau \le \frac{1}{5}\log_\sigma n$, in $O(n/\tau)$ time if T is given in a packed representation.*

As in [69], for a τ-synchronising set A, let $\mathrm{succ}_A(i) := \min\{j \in A \cup \{n - 2\tau + 2\} : j \ge i\}$.

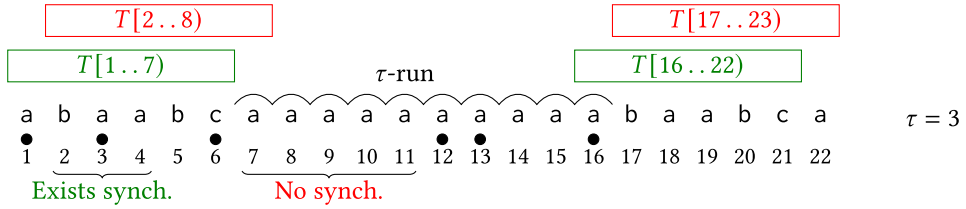


Fig. 2. An example of a τ-synchronising set ($\tau = 3$) of this string is $A = \{1, 3, 6, 12, 13, 16\}$. Fragments $T[1\,..\,7)$ and $T[16\,..\,22)$ match and thus 1, 16 are both in A. Fragments $T[2\,..\,8)$ and $T[17\,..\,23)$ match and thus 2, 17 are both *not* in A. Among every three consecutive positions (that are sufficiently far from the end of the string) there is a synchronising position, except for the positions $7, \ldots, 11$ which imply a long fragment with period $\frac{1}{3}\tau = 1$ (a so-called τ-run).

Lemma 2.6 ([69, Fact 3.2]). *If $p = per(T[i\,..\,i + 3\tau - 2]) \le \frac{1}{3}\tau$, then $T[i\,..\,succ_A(i) + 2\tau - 1)$ is the longest prefix of $T[i\,..\,|T|]$ with period p.*

Lemma 2.7 ([69, Fact 3.3]). *If a string U with $|U| \ge 3\tau - 1$ and $per(U) > \frac{1}{3}\tau$ occurs at positions i and j in T, then $succ_A(i) - i = succ_A(j) - j \le |U| - 2\tau$.*

A *τ-run* R is a run of length at least $3\tau - 1$ with period at most $\frac{1}{3}\tau$. The *Lyndon root* of R is the lexicographically smallest cyclic shift of $R[1\,..\,per(R)]$. The proof of the following lemma resembles an argument given in [69, Section 6.1.2]; it is presented in Appendix B for completeness.

Lemma 2.8. *For a positive integer τ, a string $T \in [0\,..\,\sigma)^n$ contains $O(n/\tau)$ τ-runs. Moreover, if $\tau \le \frac{1}{4}\log_\sigma n$, given a packed representation of T, we can compute all τ-runs in T and group them by their Lyndon roots in $O(n/\tau)$ time. Within the same complexities, for each τ-run, we can compute the two leftmost occurrences of its Lyndon root.*

Theorem 2.9 ([69, Theorem 4.3]). *Given a packed representation of a string $T \in [0\,..\,\sigma)^n$ and a τ-synchronising set A of T of size $O(n/\tau)$ for $\tau = O(\log_\sigma n)$, we can compute in $O(n/\tau)$ time the lexicographic order of all suffixes of T starting at positions in A.*

We often want to preprocess T to be able to answer queries of the form $\mathsf{LCP}(T[i\,..\,n], T[j\,..\,n])$. For this case, there exists an optimal data structure that is based on synchronising sets.

Theorem 2.10 ([69, Theorem 5.4]). *Given a packed representation of a string $T \in [0\,..\,\sigma)^n$, LCP queries on suffixes of T can be answered in $O(1)$ time after an $O(n/\log_\sigma n)$-time preprocessing.*

3 Sublinear-Time LCS

We provide different solutions depending on the length ℓ of an LCS. Lemmas 3.1, 3.2 and 3.8 directly yield Theorem 1.1.

3.1 Solutions for Short and Long LCS

Lemma 3.1 (Short LCS). *The LCS problem can be solved in $O(n/\log_\sigma n)$ time if $\ell \le \frac{1}{3}\log_\sigma n$.*

Proof. Let us set $m = \lfloor \frac{1}{3}\log_\sigma n \rfloor$. Given the packed representations of strings S and T, we can encode any length-$2m$ substring of the strings as a number in $[0\,..\,\lfloor n^{2/3} \rfloor)$ in $O(1)$ time. Sorting $O(n/m)$ such numbers via Counting Sort takes $O(n/m + n^{2/3})$ time. We use the so-called standard trick: we split both S and T into $O(n/m)$ fragments, each of length $2m$ (perhaps apart from the last one), starting at positions equivalent to $1 \pmod m$. For each of the strings, we obtain at most $\sigma^{2m} = O(n^{2/3})$ *distinct* substrings corresponding to these fragments. For each of the two strings, we can compute the distinct such substrings in $O(n/m + n^{2/3})$ time by sorting all considered substrings.

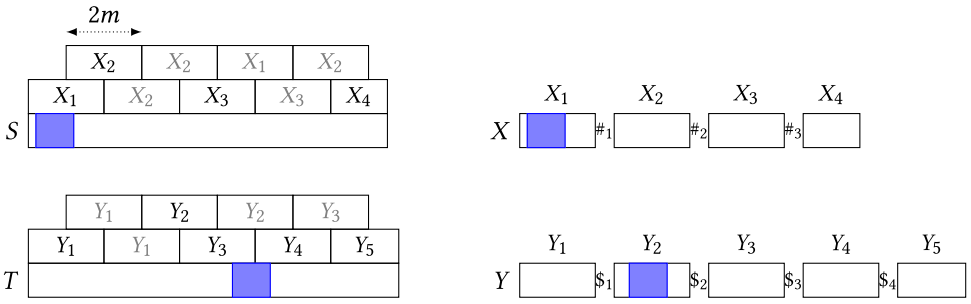


Fig. 3. Left: Covering S and T by fragments of length (up to) $2m$. All distinct substrings in S and in T are $X_1, \ldots, X_4$ and $Y_1, \ldots, Y_5$, respectively. The LCS of length m (in blue) is a substring of X_1 in S and of Y_2 in T. Right: Strings X and Y; their LCS (shown in blue) is the same as the LCS of S and T.

Let $X_1, \ldots, X_p$ and $Y_1, \ldots, Y_q$ be the resulting distinct substrings; see Figure 3. Then the problem can be reduced to computing the LCS of $X = X_1 \#_1 X_2 \#_2 \ldots \#_{p-1} X_p$ and $Y = Y_1 \$_1 Y_2 \$_2 \ldots \$_{q-1} Y_q$ for distinct letters $\#_1, \ldots, \#_{p-1}, \$_1, \ldots, \$_{q-1}$. Strings X and Y have length $O(n^{2/3}m)$ and their LCS can be computed in linear time by constructing the suffix tree of $X\#Y$, where # is a letter that does not occur in XY [52]. Thus, the overall time complexity is $O(n/m + n^{2/3}m) = O(n/\log_\sigma n)$. □

The proof of the following lemma, for the case when an LCS is long, i.e., of length $\ell = \Omega(\frac{\log^4 n}{\log^2 \sigma})$, uses difference covers and the $O(N \log N)$-time solution to the Two String Families LCP Problem. This proof closely follows [34].

Lemma 3.2 (Long LCS). *The LCS problem can be solved in $O(n/\log_\sigma n)$ time if $\ell = \Omega(\frac{\log^4 n}{\log^2 \sigma})$.*

Before proceeding to the proof, we need to introduce difference covers. We say that (D, h) is a *d-cover* if $D \subseteq \mathbb{Z}_+$ and h is a (constant-time computable) function such that for positive integers i, j we have $0 \le h(i,j) < d$ and $i + h(i,j), j + h(i,j) \in D$. The following fact synthesises a well-known construction implicitly used in [29], for example.

Theorem 3.3 ([45, 77]). *For each positive integer d there is a d-cover (D, h) such that $D \cap [1\,..\,n]$ is of size $O(\frac{n}{\sqrt{d}})$ and can be constructed in $O(\frac{n}{\sqrt{d}})$ time.*

Proof of Lemma 3.2. Let us assume that the answer to LCS is of length $\ell \ge d$ for some parameter d. We first use Theorem 3.3 to compute in $O(n/\sqrt{d})$ time a set $D = D' \cap [1\,..\,n]$ such that (D', h) is a d-cover for a function h. Let us now consider a position i from S and a position j from T such that $S[i\,..\,i+\ell) = T[j\,..\,j+\ell)$. We have that $0 \le h(i,j) < d \le \ell$, so for $i' = i + h(i,j)$, $j' = j + h(i,j)$ we have:

$$S[i\,..\,i') = T[j\,..\,j'),\ S[i'\,..\,i+\ell) = T[j'\,..\,j+\ell),\ \text{and } i', j' \in D.$$

In particular (see Figure 4 for an example):

$$\mathsf{LCP}((S[1\,..\,i'))^R, (T[1\,..\,j'))^R) + \mathsf{LCP}(S[i'\,..\,|S|], T[j'\,..\,|T|]) = \ell.$$

Hence, if we want to find an LCS whose length is at least d, we can use the elements of the d-cover as anchors between occurrences of the sought LCS, the length of which equals:

$$\max_{i,j \in D} \{\mathsf{LCP}((S[1\,..\,i))^R, (T[1\,..\,j))^R) + \mathsf{LCP}(S[i\,..\,|S|], T[j\,..\,|T|])\}. \tag{1}$$

As also done in [34], we set:

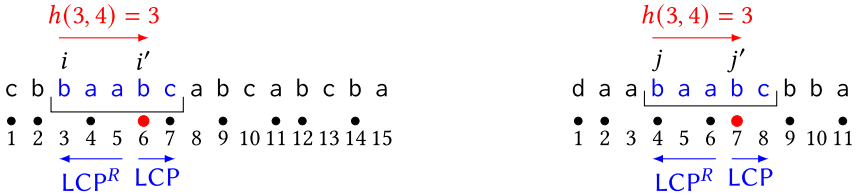


Fig. 4. Strings from Figure 1 with their LCS $S[3\,..\,8) = T[4\,..\,9)$ of length $\ell = 5$. The dots correspond to elements of a 5-cover (D, h) where $D = \{1, 2, 4, 6, 7, 9, 11, 12, 14, \ldots\}$. For $i = 3$, $j = 4$, we have $i' = i + h(i, j) = 6$, $j' = j + h(i, j) = 7$ and $\mathsf{LCP}((S[1\,..\,i'))^R, (T[1\,..\,j'))^R) + \mathsf{LCP}(S[i'\,..\,|S|], T[j'\,..\,|T|]) = \ell = 5$.

– $\mathcal{P} := \{((S[1\,..\,i))^R, S[i\,..\,|S|]) : i \in D\}$, $\mathcal{Q} := \{((T[1\,..\,i))^R, T[i\,..\,|T|]) : i \in D\}$ and
– $\mathcal{F} := \{U : (U, V) \in \mathcal{P} \cup \mathcal{Q} \text{ or } (V, U) \in \mathcal{P} \cup \mathcal{Q} \text{ for some string } V\}$.

It is then readily verified that, after building the compacted trie $\mathcal{T}(\mathcal{F})$, evaluating Equation (1) reduces to solving the Two String Families LCP Problem for $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{F}$ with $N = O(|D \cap [1\,..\,n]|) = O(n/\sqrt{d})$. In order to build $\mathcal{T}(\mathcal{F})$, due to Lemma 2.3, it suffices to sort the elements of $\mathcal{F}$ lexicographically and answer LCP queries between consecutive elements in the resulting sorted list. To this end, we first preprocess $S\#_1S^R\#_2T\#_3T^R$, where $\#_i \notin \Sigma$ for $i \in \{1, 2, 3\}$ are distinct letters, in $O(n/\log_\sigma n)$ time as in Theorem 2.10, in order to allow for $O(1)$-time LCP queries. Next, we employ merge sort in order to sort the elements of $\mathcal{F}$, performing each comparison using an LCP query. This step requires $O(|\mathcal{F}| \log |\mathcal{F}|)$ time. Then, answering LCP queries for consecutive elements requires $O(|\mathcal{F}|)$ time in total. We finally employ Lemma 1.3 to solve the instance of Two String Families LCP Problem. We have that each of $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{F}$ is of size at most $N = O(n/\sqrt{d})$. The overall time complexity is:

$$O\left(N \log N + n/\log_\sigma n\right) = O(n \log n/\sqrt{d} + n/\log_\sigma n),$$

which is $O(n/\log_\sigma n)$ if $d = \Omega(\frac{\log^4 n}{\log^2 \sigma})$. □

Remark 3.4. The only modification compared to the solution of [34] lies in the construction of $\mathcal{T}(\mathcal{F})$. In [34], $\mathcal{T}(\mathcal{F})$ is extracted from the generalised suffix tree of strings S, T, S^R and T^R, which we cannot afford to construct, as its construction requires $\Omega(n)$ time. Here we employ Theorem 2.10 instead.There are arcs, glued together, drawn above positions

3.2 Solution for Medium-Length LCS

We now give a solution to the LCS problem for ℓ such that $\frac{1}{3}\log_\sigma n \le \ell \le 2^{\sqrt{\log n}}$. We first construct three subsets of positions in $S\$T$, where $\$ \notin \Sigma$, of size $O(n/\log_\sigma n)$ as follows. For $\tau = \left\lfloor \frac{1}{9}\log_\sigma n \right\rfloor$, let A_I be a τ-synchronising set of $S\$T$. For each τ-run in $S\$T$, we insert to A_{II} the starting positions of the first two occurrences of the Lyndon root of the τ-run and to A_{III} the last position of the τ-run. The elements of A_{II} and A_{III} store the τ-run they originate from. Finally, we denote $A_j^S = A_j \cap [1\,..\,|S|]$ and $A_j^T = \{a - |S| - 1 : a \in A_j, a > |S| + 1\}$ for $j \in \{I, II, III\}$. The following lemma shows that there exists an LCS of S and T for which $A_I \cup A_{II} \cup A_{III}$ is a set of *anchors* that satisfies certain distance requirements.

Lemma 3.5. *If an LCS of S and T has length $\ell \ge 3\tau$, then there exist positions $i^S \in [1\,..\,|S|]$, $i^T \in [1\,..\,|T|]$, a shift $\delta \in [0\,..\,\ell)$ and $j \in \{I, II, III\}$ such that $S[i^S\,..\,i^S + \ell) = T[i^T\,..\,i^T + \ell)$, $i^S + \delta \in A_j^S$, $i^T + \delta \in A_j^T$, and:*

— *if* $j = I$*, then we can choose a* $\delta \in [0\,..\,\tau)$;
— *if* $j = II$*, then* $S[i^S\,..\,i^S + \ell)$ *is contained in the* τ*-run from which* $i^S + \delta \in A^S$ *originates*;
— *if* $j = III$*, then* $S[i^S\,..\,i^S + \delta]$ *is a suffix of the* τ*-run from which* $i^S + \delta \in A^S$ *originates.*

Proof. By the assumption, there exist $i^S \in [1\,..\,|S|]$ and $i^T \in [1\,..\,|T|]$ such that $S[i^S\,..\,i^S + \ell) = T[i^T\,..\,i^T + \ell)$. Let us choose any such pair (i^S, i^T) minimising the sum $i^S + i^T$. We have the following cases.

(1) If $\mathrm{per}(S[i^S\,..\,i^S + 3\tau - 2]) > \frac{1}{3}\tau$, then, by the definition of a τ-synchronising set, there exist some elements $a^S \in A^S_I \cap [i^S\,..\,i^S + \tau)$ and $a^T \in A^T_I \cap [i^T\,..\,i^T + \tau)$. Let us choose the smallest such elements. By Lemma 2.7, we have $a^S - i^S = a^T - i^T \in [0\,..\,\tau)$.
(2) Else, $p = \mathrm{per}(S[i^S\,..\,i^S + 3\tau - 2]) \le \frac{1}{3}\tau$. We have two subcases.
   (a) If $p = \mathrm{per}(S[i^S\,..\,i^S + \ell))$, then, by the choice of i^S and i^T, there exists a τ-run R_S in S that starts at position in $(i^S - p\,..\,i^S]$ and a τ-run R_T in T that starts at a position in $(i^T - p\,..\,i^T]$. Moreover, by Lemma 2.2, the Lyndon roots of the two runs are equal. For each $X \in \{S, T\}$, let us choose a^X as the leftmost starting position of a Lyndon root of R_X that is weakly to the right of i^X. We have $a^S - i^S = a^T - i^T \in [0\,..\,\lfloor \frac{1}{3}\tau \rfloor)$. Each position a^X will be the starting position of the first or the second occurrence of the Lyndon root of R_S, so $a^S \in A^S_{II}$ and $a^T \in A^T_{II}$.
   (b) Else, $p \ne \mathrm{per}(S[i^S\,..\,i^S + \ell))$. We have $d := \min\{b \ge p\,:\, S[i^S + b] \ne S[i^S + b - p]\} < \ell$ (and $d \ge 3\tau - 1$). In this case, $a^S = i^S + d - 1$ and $a^T = i^T + d - 1$ are the ending positions of τ-runs with period p in S and T, respectively, so $a^S \in A^S_{III}$ and $a^T \in A^T_{III}$.

In each case, we set $\delta := a^S - i^S = a^T - i^T$. □

The case when $j = I$ in the above lemma corresponds to the Two String Families LCP Problem with $\mathcal{P}$ and $\mathcal{Q}$ being $(\tau, 2^{\sqrt{\log n}})$-families. In this case, we use Lemma 1.5. Let us introduce a variant of the Two String Families LCP Problem that intuitively corresponds to the case when $j \in \{II, III\}$. A family of string pairs $\mathcal{P}$ is called a *prefix family* if there exists a string Y such that, for each $(U, V) \in \mathcal{P}$, U is a prefix of Y.

Lemma 3.6. *An instance of the Two String Families LCP Problem in which* $\mathcal{P} \cup \mathcal{Q}$ *is a prefix family can be solved in* $\mathcal{O}(N)$ *time.*

Proof. By traversing $\mathcal{T}(\mathcal{F})$ we can compute in $\mathcal{O}(N)$ time a list $\mathcal{R}$ being a union of sets $\mathcal{P}$ and $\mathcal{Q}$ in which the second components are ordered lexicographically.

Consider an element $e = (U, V) \in \mathcal{P}$. Let $\text{lex-pred}(e) = (Y_1, Y_2)$ be the predecessor of e in $\mathcal{R}$ that originates from $\mathcal{Q}$ and satisfies $|Y_1| \ge |U|$. If there is no such predecessor, we assume that $\text{lex-pred}(e)$ is undefined. Similarly, let $\text{lex-succ}(e) = (Z_1, Z_2)$ be the successor of e in $\mathcal{R}$ that originates from $\mathcal{Q}$ and satisfies $|Z_1| \ge |U|$ (inspect Figure 5). Further, let $\gamma(e) = \max\{\mathsf{LCP}(U, Y_1) + \mathsf{LCP}(V, Y_2), \mathsf{LCP}(U, Z_1) + \mathsf{LCP}(V, Z_2)\}$; if any of the pairs (Y_1, Y_2), (Z_1, Z_2) is undefined, we set the corresponding component of the max operation to 0. We also define the same notations for $e \in \mathcal{Q}$ with $\mathcal{P}$ and $\mathcal{Q}$ swapped.

Claim 3.7. $\mathrm{maxPairLCP}(\mathcal{P}, \mathcal{Q}) = \max_{e \in \mathcal{P} \cup \mathcal{Q}} \gamma(e)$.

Proof. First, we clearly have $\mathrm{maxPairLCP}(\mathcal{P}, \mathcal{Q}) \ge \max_{e \in \mathcal{P} \cup \mathcal{Q}} \gamma(e)$.

Let $(P_1, P_2) \in \mathcal{P}$ and $(Q_1, Q_2) \in \mathcal{Q}$ be such that $\mathsf{LCP}(P_1, Q_1) + \mathsf{LCP}(P_2, Q_2) = \mathrm{maxPairLCP}(\mathcal{P}, \mathcal{Q})$. Without loss of generality, let us assume that $|P_1| \le |Q_1|$. We further assume that (Q_1, Q_2) precedes (P_1, P_2) in $\mathcal{R}$.

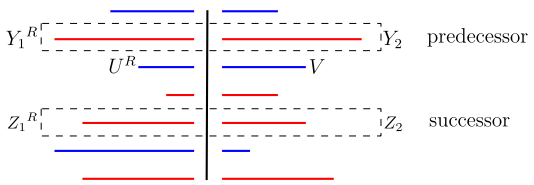


Fig. 5. The setting in Lemma 3.6 on list $\mathcal{R}$. With red colour, we denote the elements of $\mathcal{P}$ and with blue colour the elements of $\mathcal{Q}$. For element $e = (U, V)$ from $\mathcal{Q}$, we have lex-pred$(e) = (Y_1, Y_2)$ and lex-succ$(e) = (Z_1, Z_2)$.

Let $(Y_1, Y_2) =$ lex-pred$((P_1, P_2))$. Now, we have $\mathsf{LCP}(P_1, Y_1) = |P_1| \geq \mathsf{LCP}(P_1, Q_1)$ since $|P_1| \leq |Y_1|$ and P_1 and Y_1 are prefixes of the same string (as $\mathcal{P} \cup \mathcal{Q}$ is a prefix family). By Fact 2.1 we have $\mathsf{LCP}(P_2, Y_2) \geq \mathsf{LCP}(P_2, Q_2)$ since (Q_1, Q_2), (Y_1, Y_2) and (P_1, P_2) appear in $\mathcal{R}$ in this order. Thus:

$$\text{maxPairLCP}(\mathcal{P}, \mathcal{Q}) = \mathsf{LCP}(P_1, Q_1) + \mathsf{LCP}(P_2, Q_2) \leq \mathsf{LCP}(P_1, Y_1) + \mathsf{LCP}(P_2, Y_2) \leq \max_{e \in \mathcal{P} \cup \mathcal{Q}} \gamma(e).$$

The case when (P_1, P_2) precedes (Q_1, Q_2) in $\mathcal{R}$ is symmetric. □

To compute lex-succ(e) and lex-pred(e) for each $e \in \mathcal{P} \cup \mathcal{Q}$ we proceed as follows. We process the list $\mathcal{R}$ in the order of non-decreasing lengths of the first components. This order can be computed in $\mathcal{O}(N)$ time using the compacted trie $\mathcal{T}(\mathcal{F})$. After processing all elements e of some length, we remove them from $\mathcal{R}$.

We maintain the list $\mathcal{R}$ using the data structure of Gabow and Tarjan [57] for a special case of the union-find problem, in which the elements correspond to numbers in $[1\,..\,|\mathcal{R}|]$ representing the subsequent elements of the initial list $\mathcal{R}$ and the sets are formed by consecutive integers. Let us think of elements of $\mathcal{R}$ originating from $\mathcal{P}$ to be coloured by red and elements originating from $\mathcal{Q}$ to be coloured by blue. The sets the union-find data structure is initialised with correspond to maximal sequences of elements of the same colour in $\mathcal{R}$. Each set has as an id its smallest element. Each set also maintains the following satellite information: pointers to the head and to the tail of a list of all non-deleted elements in this set. Every element of $\mathcal{R}$ stores a pointer to the element in the list in which it is represented. It can thus be deleted at any moment in $\mathcal{O}(1)$ time. When the satellite list of a set S_i becomes empty, we union S_{i-1}, S_i and S_{i+1} (provided that the respective sets exist). To find lex-pred(e) we perform the following; the procedure of lex-succ(e) is analogous. We find the set S_i to which e belongs using a find operation. Let the id of S_i be α. By using another find operation to find $\alpha - 1$, we find the set S_{i-1}. The tail of the list of S_{i-1} is lex-pred(e); in the analogous procedure, the head of the list of S_{i+1} is lex-succ(e). The algorithm is correct because we process $\mathcal{R}$ in the order of non-decreasing lengths. Each union or find operation requires $\mathcal{O}(1)$ amortised time [57]. Thus, this procedure takes $\mathcal{O}(N)$ time in total.

Finally, we preprocess the compacted trie $\mathcal{T}(\mathcal{F})$ in $\mathcal{O}(N)$ time to be able to answer lowest common ancestor (LCA) queries in $\mathcal{O}(1)$ time [21]. For any $e = (U, V)$, given lex-pred(e) and lex-succ(e) we can compute $\gamma(e)$ in $\mathcal{O}(1)$ time by answering LCP queries using the LCA data structure. □

We are now ready to state the main result of this subsection. The proof of Lemma 1.5 is deferred to Section 4.

Lemma 3.8 (Medium-Length LCS). *The LCS problem can be solved in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time using $\mathcal{O}(n/\log_\sigma n)$ space if $\frac{1}{3} \log_\sigma n \leq \ell \leq 2^{\sqrt{\log n}}$.*

Proof. Recall that $\tau = \left\lfloor \frac{1}{9} \log_\sigma n \right\rfloor$. The set of anchors $A = A_I \cup A_{II} \cup A_{III}$ consists of a τ-synchronising set and of $\mathcal{O}(1)$ positions per each τ-run in $S\$T$. Hence, $|A| = \mathcal{O}(n/\tau)$ and A can be

constructed in $O(n/\tau)$ time by Theorem 2.5 and 2.8. We also use Lemma 2.8 to group all τ-runs by their Lyndon roots.

We construct sets of pairs of substrings of $X = STS^RT^R$. First, for $\Delta = \left\lfloor 2^{\sqrt{\log n}} \right\rfloor$:

$$\mathcal{P}_I = \{((S[a-\tau\,..\,a))^R, S[a\,..\,a+\Delta)) \;:\; a \in A_I^S\}.$$

Then, for each group $\mathcal{G}$ of τ-runs in S and T with equal Lyndon root, we construct the following set of string pairs:

$$\mathcal{P}_{II}^{\mathcal{G}} = \{((S[x\,..\,a))^R, S[a\,..\,y]) \;:\; a \in A_{II}^S \text{ that originates from } \tau\text{-run } S[x\,..\,y] \in \mathcal{G}\}.$$

We define the *tail* of a τ-run $S[x\,..\,y]$ with period p and Lyndon root $S[x'\,..\,x'+p)$ as $(y+1-x') \bmod p$ (and same for τ-runs in T). For each group of τ-runs in S and T with equal Lyndon roots, we group the τ-runs belonging to it by their tails. This can be done in $O(n/\tau)$ time using Radix Sort since the tail values are up to $\frac{1}{3}\tau$. For each group $\mathcal{G}$ of τ-runs in S and T with equal Lyndon root and tail, we construct the following set of string pairs:

$$\mathcal{P}_{III}^{\mathcal{G}} = \{((S[x\,..\,y))^R, S[y\,..\,|S|]) \;:\; S[x\,..\,y] \in \mathcal{G}\}.$$

Simultaneously, we create sets $\mathcal{Q}_I$, $\mathcal{Q}_{II}^{\mathcal{G}}$ and $\mathcal{Q}_{III}^{\mathcal{G}}$ defined with T instead of S.

By Lemma 3.5, it suffices to output the maximum of $\text{maxPairLCP}(\mathcal{P}_I, \mathcal{Q}_I)$, $\text{maxPairLCP}(\mathcal{P}_{II}^{\mathcal{G}}, \mathcal{Q}_{II}^{\mathcal{G}})$ and $\text{maxPairLCP}(\mathcal{P}_{III}^{\mathcal{G}}, \mathcal{Q}_{III}^{\mathcal{G}})$, where $\mathcal{G}$ ranges over groups of τ-runs in $S\$T$.

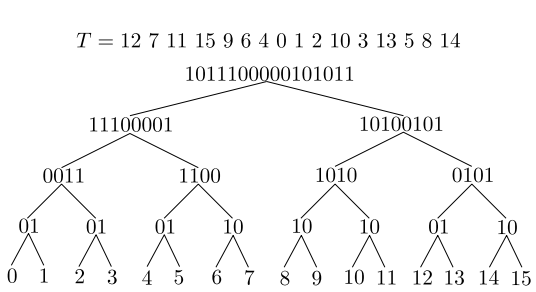
Computing any individual value of maxPairLCP can be expressed as an instance of the Two String Families LCP Problem provided that all the first and second components of families are represented as nodes of compacted tries. We will use Lemma 2.3 to construct these compacted tries. LCP queries can be answered efficiently due to Theorem 2.10, so it suffices to be able to sort all the first and second components of each pair of string pair sets lexicographically. Each of the sets $\mathcal{P}_I$ and $\mathcal{Q}_I$ can be ordered by the second components using Theorem 2.9 since A_I is a τ-synchronising set, and by the first components via a straightforward tabulation approach because the number of possible τ-length strings is $\sigma^\tau = O(n^{1/9})$. In a set $\mathcal{P}_{II}^{\mathcal{G}}$, all first and all second components are prefixes of a single string (a power of the common Lyndon root). Hence, they can be sorted simply by comparing their lengths. This sorting is performed simultaneously for all the families $\mathcal{P}_{II}^{\mathcal{G}}$, $\mathcal{Q}_{II}^{\mathcal{G}}$ in $O(n/\tau)$ time via Radix Sort. Finally, to sort the second components of the sets $\mathcal{P}_{III}^{\mathcal{G}}$ $\mathcal{Q}_{III}^{\mathcal{G}}$, instead of comparing strings of the form $S[y\,..\,|S|]$ (and same for T), we can equivalently compare strings $S[y-2\tau+1\,..\,|S|]$ which are known to start at positions from a τ-synchronising set by Lemma 2.6. This sorting is done across all groups using Radix Sort and Theorem 2.9.

Finally, we observe that $(\mathcal{P}_I, \mathcal{Q}_I)$ is a (τ, Δ)-family of size $N = O(n/\tau)$, and thus the value $\text{maxPairLCP}(\mathcal{P}_I, \mathcal{Q}_I)$ can be computed in $O(n \log \sigma / \sqrt{\log n})$ time and $O(n/\log_\sigma n)$ space using Lemma 1.5. On the other hand, $(\mathcal{P}_{II}^{\mathcal{G}}, \mathcal{Q}_{II}^{\mathcal{G}})$ and $(\mathcal{P}_{III}^{\mathcal{G}}, \mathcal{Q}_{III}^{\mathcal{G}})$ are prefix families of total size $O(n/\tau)$, so the corresponding instances of the Two String Families LCP Problem can be solved in $O(n/\log_\sigma n)$ total time using Lemma 3.6. □

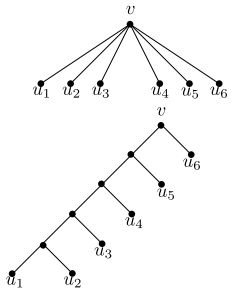
4 Proof of Lemma 1.5—Solution to a Special Case of the Two String Families LCP Problem via Wavelet Trees

Wavelet Trees. For an arbitrary alphabet Σ, the *skeleton tree* for Σ is a full binary tree $\mathcal{T}$ together with a bijection between Σ and the leaves of $\mathcal{T}$. For a node $v \in \mathcal{T}$, let Σ_v denote the subset of Σ that corresponds to the leaves in the subtree of v.

For a skeleton tree $\mathcal{T}$ and a string $T \in \Sigma^*$, the *$\mathcal{T}$-shaped wavelet tree* of T is $\mathcal{T}$ augmented with bit-vectors assigned to its internal nodes (inspect Figure 6(a) for a wavelet tree with a 'standard'



(a) Standard skeleton tree with $|T| = \sigma = 16$.



(b) Binarisation with height $O(\alpha + \log N)$.

Fig. 6. (a) Let v be left child of the root node. Then $\Sigma_v = \{0, 1, \ldots, 7\}$, $T_v = 7, 6, 4, 0, 1, 2, 3, 5$ and so $B_v = 11100001$: $7, 6, 4, 5$ belong to the right subtree of v and $0, 1, 2, 3$ to the left. (b) For each i, let the size of the subtree rooted at u_i be 2^i. The binarisation from [9] leads to height $O(\alpha + \log N)$, favouring heavier children.

skeleton tree). For an internal node v with left child v_L and right child v_R, let T_v denote the maximal subsequence of T that consists of letters from Σ_v; the bit-vector $B_v[1\,..\,|T_v|]$ is defined so that $B_v[i] = 0$ if $T_v[i] \in \Sigma_{v_L}$ and $B_v[i] = 1$ if $T_v[i] \in \Sigma_{v_R}$.

Wavelet trees were introduced in [62]. One can derive an $O(n \log \sigma)$-time construction algorithm directly from their definition; more efficient construction algorithms were presented in [9, 83].

Theorem 4.1 (See [83, Theorem 2]). *Given the packed representation of a string $T \in [0\,..\,\sigma)^n$ and a skeleton tree $\mathcal{T}$ of height h, the $\mathcal{T}$-shaped wavelet tree of T takes $O(nh/\log n + \sigma)$ space and can be constructed in $O(nh/\sqrt{\log n} + \sigma)$ time.*

Wavelet trees are sometimes constructed for sequences $T \in \mathcal{M}^*$ over an alphabet $\mathcal{M} \subseteq \Sigma^*$ that itself consists of strings (see e.g., [69]). In this case, the skeleton tree $\mathcal{T}$ is often chosen to resemble the compacted trie of $\mathcal{M}$. Formally, we say that a skeleton tree $\mathcal{T}$ for some $\mathcal{M} \subseteq \Sigma^*$ is *prefix-consistent* if each node $v \in \mathcal{T}$ admits a label $\mathsf{val}(v) \in \Sigma^*$ such that:

- the restriction of the function $\mathsf{val}(v)$ to the leaves $v \in \mathcal{T}$ is a bijection from those leaves to $\mathcal{M}$;
- if v is a node with children v_L, v_R, then, for all leaves u_L, u_R in the subtrees of v_L and v_R, respectively, the string $\mathsf{val}(v)$ is the longest common prefix of $\mathsf{val}(u_L)$ and $\mathsf{val}(u_R)$.

We observe that if $\mathcal{M} \subseteq \{0, 1\}^\alpha$ for some integer α, then the compacted trie $\mathcal{T}(\mathcal{M})$ is a prefix-consistent skeleton tree for $\mathcal{M}$. For larger alphabets, we binarise $\mathcal{T}(\mathcal{M})$ as follows.

Lemma 4.2. *Given the compacted trie $\mathcal{T}(\mathcal{M})$ of a set $\mathcal{M} \subseteq \Sigma^\alpha$, a prefix-consistent skeleton tree for $\mathcal{M}$ of height $O(\alpha + \log |\mathcal{M}|)$ can be constructed in $O(|\mathcal{M}|)$ time, with each node v associated to a node v' of $\mathcal{T}(\mathcal{M})$ such that $\mathsf{val}(v) = \mathsf{val}(v')$.*

Proof. We use [9, Corollary 3.2], where the authors showed that any rooted tree with m leaves and of height h can be binarised in $O(m)$ time so that the resulting tree is of height $O(h + \log m)$. For $\mathcal{T}(\mathcal{M})$, we obtain height $O(\alpha + \log |\mathcal{M}|)$ and time $O(|\mathcal{M}|)$ (inspect Figure 6(b)). □

Lemma 1.5. *An instance of the Two String Families LCP Problem in which $\mathcal{P}$ and $\mathcal{Q}$ are (α, β)-families can be solved in time $O(N(\alpha + \log N)(\log \beta + \sqrt{\log N})/\log N)$ and space $O(N + N\alpha/\log N)$.*

Proof. If $\log \beta = \Omega(\log N)$, the stated bounds are not better than those of Lemma 1.3. We henceforth assume that $\log \beta = o(\log N)$.

Let $\mathcal{R}$ be the list obtained by sorting $\mathcal{P} \cup \mathcal{Q}$ with respect to the lexicographic order of the second components. For any sublist $\mathcal{X} = (U_1, V_1), \ldots, (U_m, V_m)$ of $\mathcal{R}$, let $\mathsf{LCPs}(\mathcal{X})$ denote the

Algorithm 1: Computation of LCPs Lists

$\mu_0 := \mu_1 := \mathcal{L}.\texttt{read}();$
while not $B.\texttt{end\_of\_stream}()$ **do**
  $b := B.\texttt{read}();$
  $G_b.\texttt{write}(G.\texttt{read}());$
  $\mathcal{L}_b.\texttt{write}(\mu_b);$
  $\mu_b := \mathcal{L}.\texttt{read}();$
  $\mu_{1-b} := \min(\mu_{1-b}, \mu_b);$

representation of the list $0, \mathsf{LCP}(V_1, V_2), \ldots, \mathsf{LCP}(V_{m-1}, V_m)$ as a packed string over alphabet $[0\,..\,\beta]$ in space $O(1 + N/\log_\beta N)$. For each node v of the wavelet tree, let $\mathcal{R}_v$ be the sublist of $\mathcal{R}$ composed of elements whose first component is in the leaf list Σ_v of v.

Claim 4.3. Consider an instance of the Two String Families LCP Problem in which $\mathcal{P}$ and $\mathcal{Q}$ are (α, β)-families with $\log \beta = o(\log N)$. We can construct, in $O(N(\alpha + \log N)/\sqrt{\log N})$ time and $O(N + N\alpha/\log N)$ space, a wavelet tree of height $O(\alpha + \log N)$ for the first components of $\mathcal{R}$ (some possibly padded with a $\$ \notin \Sigma$).

Moreover, in $O(N(\alpha + \log N)\log\beta/\log N)$ time and $O(N)$ space, we can compute a bit-vector G_v specifying the origin ($\mathcal{P}$ or $\mathcal{Q}$) of each element of $\mathcal{R}_v$ and the list $\mathcal{L}_v = \mathsf{LCPs}(\mathcal{R}_v)$ for each node v of the wavelet tree in the BFS order, such that after computing G_u and $\mathcal{L}_u$ for each child u of a node v, G_v and $\mathcal{L}_v$ are deleted.

Proof. By traversing $\mathcal{T}(\mathcal{F})$ we can sort in $O(N)$ time all elements of $\mathcal{P} \cup \mathcal{Q}$ by the second components, obtaining the list $\mathcal{R}$. We also store a bit-vector G of length $|\mathcal{R}|$ that specifies, for each element of $\mathcal{R}$, which of the sets $\mathcal{P}, \mathcal{Q}$ it originates from. We then construct the prefix-consistent skeleton tree for the sequence of strings being the first components of pairs from $\mathcal{R}$ with Lemma 4.2 and use it to construct the wavelet tree of the same sequence using Theorem 4.1. Before said skeleton and wavelet trees are constructed, we pad each string with a letter $\$ \notin \Sigma$ to make them all of length α; in what follows, we ignore the nodes of the wavelet tree whose path-label contains a \$.

The list $\mathsf{LCPs}(\mathcal{R})$ can be computed in $O(N)$ time when constructing $\mathcal{R}$. For each node v of the wavelet tree, we wish to compute $\mathcal{L}_v = \mathsf{LCPs}(\mathcal{R}_v)$ and the corresponding bit-vector G_v of origins ($\mathcal{P}$ or $\mathcal{Q}$) of the elements of $\mathcal{R}_v$. We will construct the lists $\mathsf{LCPs}(\mathcal{R}_v)$ without actually computing $\mathcal{R}_v$. We process the nodes in the BFS order.

Computation of LCPs *lists.* We apply Algorithm 1, that we describe below, to each node v of the wavelet tree, with $\mathcal{L}$ being a stream representing the LCPs list $\mathcal{L}_v$, G a stream of the corresponding bit-vector of origins G_v, and B a stream of the bit-vector B_v stored in the wavelet tree node. The algorithm outputs streams $\mathcal{L}_0$ and $\mathcal{L}_1$ representing the lists $\mathcal{L}_{u_L}$ and $\mathcal{L}_{u_R}$ of the left and right children u_L, u_R of v and, analogously, streams G_0 and G_1 representing G_{u_L} and G_{u_R}.

Let us describe the algorithm and argue for its correctness. Variable b is used to store each subsequent bit of the stream B. It is then used to forward the next bit of the stream of origins G to the respective output stream.

Variables μ_0 and μ_1 are used for processing the LCPs list. Let $\mathcal{R}_v = (U_1, V_1), \ldots, (U_m, V_m)$ and assume V_0 is a sentinel empty string. We next show that before the step of the while-loop in which the ith bit of B is read, we have:

$$\mu_b = \mathsf{LCP}(V_{p_b}, V_i), \text{ where } p_b = \max(\{0\} \cup \{p \in [1\,..\,i) \,:\, B_v[p] = b\}). \tag{2}$$

In particular, if $p_b = 0$, i.e., no b-bit in B was encountered yet, then $\mu_b = 0$.

Algorithm 2: Application of LCPs Lists

```
r := g' := 0;
while not G.end_of_stream() do
    g := G.read();
    if g ≠ g' then r := max(r, L.read());
    else L.read();
    g' := g;
return r;
```

Let us recall that the list $\mathcal{L}_v$ starts with a 0, so initially $\mu_0 = \mu_1 = 0$ satisfy Equation (2). Let us consider the ith step of the loop with $b = B[i]$. If $p_b = 0$, then this is the first b-bit in B, so $\mu_b = 0$ is correctly output to $\mathcal{L}_b$. Otherwise, $\mu_b = \mathsf{LCP}(V_{p_b}, V_i)$ is output to $\mathcal{L}_b$, as expected. Next, μ_b becomes $\mathcal{L}_v[i+1] = \mathsf{LCP}(V_i, V_{i+1})$ which is correct since when i is incremented, $p_b = i$. Finally, μ_{1-b} becomes $\min(\mathsf{LCP}(V_{p_{1-b}}, V_i), \mathsf{LCP}(V_i, V_{i+1}))$, which is $\mathsf{LCP}(V_{p_{1-b}}, V_{i+1})$ by Fact 2.1 and the fact that the strings $V_1, \ldots, V_m$ are ordered lexicographically. When $i = m$, the next bit on stream $\mathcal{L}$ is not defined, which is not an issue as then the lines setting μ_0 and μ_1 are not relevant anymore.

Analysis. We analyse the complexity of Algorithm 1 using Proposition 2.4. The algorithm works on $O(1)$ data streams, uses $s = O(\log(\beta+1))$ bits of space and among every $t = O(1)$ instructions performs a read or write. For a node v of the wavelet tree, the total size of input and output streams in bits is proportional to the number of bits in $\mathcal{L}_v$, i.e., $O(|B_v| \log \beta)$. By Proposition 2.4, for $\tau = \frac{1}{2}\log N$, after $o(N)$ preprocessing (as $\log\beta = o(\log N)$), the algorithm can be executed in $O(1 + |B_v| \log\beta/\log N)$ time. Whenever Algorithm 1 is applied, the input streams are removed from memory. This guarantees that the additional space required to store the streams for the at most N topmost nodes that have not been processed yet is bounded by $O(N)$.

The wavelet tree, which is of height $h = O(\alpha + \log N)$, can be built in $O(Nh/\sqrt{\log N} + N)$ time using space $O(Nh/\log N + N)$ by Theorem 4.1. Over all nodes, Algorithm 1 consumes $O(Nh\log\beta/\log N)$ time and $O(N)$ space. The overall complexities follow. □

Application of LCPs *lists.* For each node v of the wavelet tree, let us define $f(v)$ as the sum of the string-depth $|\mathsf{val}(v)|$ and the maximum LCP between second components of pairs in $\mathcal{R}_v$ of different origins. For each $(U,V) \in \mathcal{P}$ and $(U',V') \in \mathcal{Q}$ such that $\mathsf{LCP}(U,U') = d$, the pairs (U,V) and (U',V') are in the same list $\mathcal{R}_v$ of one node v per each depth in $[0\,..\,d]$. Thus, to compute $\mathrm{maxPairLCP}(\mathcal{P},\mathcal{Q})$, it suffices to compute $f(v)$ for each node v of the wavelet tree and return the maximum of such values.

Greedily, again by Fact 2.1, the maximum LCP is obtained by two *consecutive* elements in the list $\mathcal{R}_v$, i.e., as $\max\{\mathcal{L}_v[i] \,:\, G_v[i] \neq G_v[i-1],\ i \in [2\,..\,|\mathcal{L}_v|]\}$. This maximum LCP is computed in Algorithm 2 exactly from this formula with $\mathcal{L}$ and G being streams representing $\mathcal{L}_v$ and G_v, as before. The algorithm is used whenever the lists $\mathcal{L}_v$ and G_v have been computed for a node v in Claim 4.3.

Let $\mathcal{R}_v = (U_1,V_1),\ldots,(U_m,V_m)$. We next show that before the ith step of the while-loop:

$$r = \max(\{0\} \cup \{\mathsf{LCP}(V_{j-1},V_j) \,:\, j \in [2\,..\,i-1],\ G_v[j-1] \neq G_v[j]\}).$$

Moreover, $g' = G_v[i-1]$ if $i > 1$ and $g' = 0$ if $i = 1$.

Let us consider the ith step of the loop with $g = G[i]$. If $i = 1$, then $\mathcal{L}[i] = 0$ by definition, so the value of r does not change regardless of the result of comparing g and g'. If $i > 1$, we have $\mathcal{L}[i] = \mathsf{LCP}(V_{i-1},V_i)$. Therefore, if $g \neq g'$, i.e., $G[i-1] \neq G[i]$, variable r is updated correctly, and otherwise the value $\mathcal{L}[i]$ can be discarded. Finally, g' becomes g.

The complexities of Algorithm 2 are asymptotically dominated by those of Algorithm 1 and hence the result follows. □

5 Sublinear-Time LCS of Several Strings

We show how to modify the results from Sections 3 and 4 to compute the LCS of λ strings $T^{(1)}, \ldots, T^{(\lambda)}$ of total length n in $O(n \log \sigma / \sqrt{\log n})$ time if $\lambda = O(\sqrt{\log n}/\log\log n)$; in particular, for $\lambda = O(1)$.

By ℓ we denote the length of an LCS. As before, the solution is divided into three cases, depending on whether ℓ is small (Section 5.3), large (Section 5.4) or medium (Section 5.5). Instead of instances of Two String Families LCP Problem, we will obtain instances of its generalisation called λ String Families LCP Problem:

> λ String Families LCP Problem
> *Input*: A compacted trie $\mathcal{T}(\mathcal{F})$ of $\mathcal{F} \subseteq \Sigma^*$ and λ non-empty sets $\mathcal{P}^{(1)}, \ldots, \mathcal{P}^{(\lambda)} \subseteq \mathcal{F}^2$, with $|\mathcal{F}| + \sum_{t=1}^{\lambda} |\mathcal{P}^{(t)}| = N$.
> *Output*: $\mathrm{maxMultiLCP}(\mathcal{P}^{(1)}, \ldots, \mathcal{P}^{(\lambda)}) = \max\{\mathsf{LCP}(P_1^{(1)}, \ldots, P_1^{(\lambda)}) + \mathsf{LCP}(P_2^{(1)}, \ldots, P_2^{(\lambda)}) : (P_1^{(t)}, P_2^{(t)}) \in \mathcal{P}^{(t)} \text{ for all } t \in [1\,..\,\lambda]\}$.

Before we proceed with the solution, in Section 5.1 we provide generalisations of some of the tools used for two input strings to the setting with multiple input strings. We also discuss efficient predecessor/successor data structures (Section 5.2).

5.1 Toolbox for Handling Multiple Input Strings

Let us recall that (D, h) is a (λ, d)-*cover* if $D \subseteq \mathbb{Z}_+$ is a set and $h : \mathbb{Z}_+^{\lambda} \to [0\,..\,d)$ is an efficiently computable function such that, for any $i_1, \ldots, i_\lambda \in \mathbb{Z}_+$ and $t \in [1\,..\,\lambda]$, we have $i_t + h(i_1, \ldots, i_\lambda) \in D$. We show an efficient construction of (λ, d)-covers; the theorem below is restated for convenience.

Theorem 1.6. *For every positive integers $\lambda \geq 2$ and d, there is a (λ, d)-cover (D, h) such that the set $D \cap [1\,..\,n]$ is of size $O(\lambda \cdot n/\sqrt[\lambda]{d})$. After $O(\log d)$-time initialisation, the set $D \cap [1\,..\,n]$ can be constructed in $O(\lambda \cdot n/\sqrt[\lambda]{d})$ time and the function $h(i_1, \ldots, h_\lambda)$ can be evaluated in $O(\lambda)$ time.*

Proof. Let $c = \left\lfloor \sqrt[\lambda]{d} \right\rfloor$. Our initialisation procedure determines c in $O(\lambda \cdot \log c) = O(\log d)$ time using exponential search.

We define:

$$D = \bigcup_{t=1}^{\lambda} \{j \in \mathbb{Z}_+ : j \bmod c^t \geq c^t - c^{t-1}\}.$$

Note that $|D \cap [1\,..\,n]| \leq \sum_{t=1}^{\lambda} |\{j \in [1\,..\,n] : j \bmod c^t \geq c^t - c^{t-1}\}| \leq \lambda \cdot n/c = O(\lambda \cdot n/\sqrt[\lambda]{d})$. Moreover, the set $D \cap [1\,..\,n]$ can be easily constructed in $O(\lambda \cdot n/\sqrt[\lambda]{d})$ time.

For any $i_1, \ldots, i_\lambda \in \mathbb{Z}_+$, we define $h(i_1, \ldots, i_\lambda) = h_\lambda$ based on a sequence $0 = h_0 \leq \cdots \leq h_\lambda$ specified using the following recursion for $t \in [1\,..\,\lambda]$:

$$\begin{aligned} h_t &= h_{t-1} + c^{t-1} \cdot \left(c - 1 - \left\lfloor \frac{i_t + h_{t-1}}{c^{t-1}} \right\rfloor \bmod c\right) \\ &= h_{t-1} + c^t - c^{t-1} + (i_t + h_{t-1}) \bmod c^{t-1} - (i_t + h_{t-1}) \bmod c^t. \end{aligned}$$

Observe that the values $(h_0, \ldots, h_\lambda)$ can be computed in $O(\lambda)$ time along with the values $(c^0, \ldots, c^\lambda)$.

A simple inductive argument proves that $h_t \in [0\,..\,c^t)$ holds for every $t \in [0\,..\,\lambda]$ and, in particular, $h_\lambda \in [0\,..\,c^\lambda) \subseteq [0\,..\,d)$. It remains to prove that $i_t + h_\lambda \in D$ holds for every $t \in [1\,..\,\lambda]$. For this, observe that:

$$\begin{aligned} i_t + h_t &= i_t + h_{t-1} + c^t - c^{t-1} + (i_t + h_{t-1}) \bmod c^{t-1} - (i_t + h_{t-1}) \bmod c^t \\ &\equiv c^t - c^{t-1} + (i_t + h_{t-1}) \bmod c^{t-1} \pmod{c^t}. \end{aligned}$$

Since $h_t \equiv h_{t+1} \equiv \cdots \equiv h_\lambda \pmod{c^t}$, we conclude that $(i_t + h_\lambda) \bmod c^t = (i_t + h_t) \bmod c^t \in [c^t - c^{t-1}\,..\,c^t)$, and thus $i_t + h_\lambda \in D$ holds as claimed. □

Let $\mathcal{L}$ be a multiset of strings and assume that each string $S \in \mathcal{L}$ is assigned a colour $\mathsf{col}(S) \in [1\,..\,\lambda]$. Note that, in general, multiple strings from $\mathcal{L}$ may be assigned the same colour. Let \$, # be auxiliary letters that are smaller and greater than all other letters of the alphabet, respectively. For a string U and a colour $c \in [1\,..\,\lambda]$, by $\mathsf{next}_c(U, \mathcal{L})$ we denote the lexicographically smallest string $S \in \mathcal{L}$ such that $S \geq U$ and $\mathsf{col}(S) = c$. If no such string S exists, we assume that $\mathsf{next}_c(U, \mathcal{L}) = \#$. Symmetrically, by $\mathsf{prev}_c(U, \mathcal{L})$ we denote the lexicographically greatest string $S \in \mathcal{L}$ such that $S \leq U$ and $\mathsf{col}(S) = c$, or \$ if no such string S exists. We also denote:

$$\begin{aligned} \mathsf{prev}(U, \mathcal{L}) &= \min\{\mathsf{prev}_c(U, \mathcal{L}) \,:\, c \in [1\,..\,\lambda]\}, \\ \mathsf{next}(U, \mathcal{L}) &= \max\{\mathsf{next}_c(U, \mathcal{L}) \,:\, c \in [1\,..\,\lambda]\}. \end{aligned}$$

A subset $\mathcal{X}$ of $\mathcal{L}$ is called a *λ-rainbow* if $|\mathcal{X}| = \lambda$ and all strings in $\mathcal{X}$ have different colours.

The next lemma can be viewed as an extension of Fact 2.1.

Lemma 5.1. *Let $\mathcal{L}$ be a multiset of strings and assume that each string $S \in \mathcal{L}$ is assigned a colour $\mathsf{col}(S) \in [1\,..\,\lambda]$ so that no two equal strings in $\mathcal{L}$ have the same colour and each colour in $[1\,..\,\lambda]$ is present in $\mathcal{L}$.*

(a) *The maximum* LCP *over all λ-rainbows $\mathcal{X} \subseteq \mathcal{L}$ equals* $\max_{S \in \mathcal{L}} \mathsf{LCP}(S, \mathsf{prev}(S, \mathcal{L}))$.
(b) *Assume that $S \in \mathcal{L}$. Let $c_1, \ldots, c_\lambda$ be a permutation of $[1\,..\,\lambda]$ such that $\mathsf{prev}_{c_1}(S, \mathcal{L}) \leq \cdots \leq \mathsf{prev}_{c_\lambda}(S, \mathcal{L})$ and $\mathsf{col}(S) = c_\lambda$. Let us denote $c_0 = c_\lambda$. Then, the maximum* LCP *of a λ-rainbow $\mathcal{X} \subseteq \mathcal{L}$ such that $S \in \mathcal{X}$ equals*:

$$\mathrm{maxLCP}(S, \mathcal{L}) := \max_{i=1}^{\lambda} \mathsf{LCP}(\mathsf{prev}_{c_i}(S, \mathcal{L}),\ \max\{\mathsf{next}_{c_j}(S, \mathcal{L}) \,:\, j \in [0\,..\,i)\}). \tag{3}$$

Proof. Proof of part (b): Let us fix $S \in \mathcal{L}$, let $\zeta = \mathrm{maxLCP}(S, \mathcal{L})$, and assume that the maximum in formula (3) is attained for $i = i'$.

Let $\mathcal{X}' \subseteq \mathcal{L}$ be a λ-rainbow such that $S \in \mathcal{X}'$ and $\mathsf{LCP}(\mathcal{X}')$ is maximal among such multisets. Among possibly many multisets $\mathcal{X}'$ that satisfy the requirements, we select the one with the largest $\min \mathcal{X}'$. By Fact 2.1, $\mathsf{LCP}(\mathcal{X}') = \mathsf{LCP}(\min \mathcal{X}', \max \mathcal{X}')$. We will prove that $\zeta = \mathsf{LCP}(\mathcal{X}')$ by showing two inequalities.

$\zeta \leq \mathsf{LCP}(\mathcal{X}')$: Since $\mathsf{LCP}(\mathcal{X}') \geq 0$ holds trivially, we can assume $\zeta > 0$. It suffices to construct a λ-rainbow $\mathcal{X} \subseteq \mathcal{L}$ such that $S \in \mathcal{X}$ and $\mathsf{LCP}(\mathcal{X}) \geq \zeta$. The required set is:

$$\mathcal{X} = \{\mathsf{prev}_{c_i}(S, \mathcal{L}) \,:\, i \in [i'\,..\,\lambda]\} \cup \{\mathsf{next}_{c_i}(S, \mathcal{L}) \,:\, i \in [1\,..\,i')\}.$$

Indeed, $S = \mathsf{prev}_{c_\lambda}(S, \mathcal{L}) \in \mathcal{X}$. Moreover, we have $\mathsf{col}(\mathsf{prev}_{c_i}(S, \mathcal{L})) = c_i$ for $i \in [i'\,..\,\lambda]$ and $\mathsf{col}(\mathsf{next}_{c_j}(S, \mathcal{L})) = c_j$ for all $j \in [1\,..\,i')$; this is because $\zeta > 0$, so $\mathsf{prev}_{c_{i'}}(S, \mathcal{L}) \neq \$$ and $\max_{j \in [1\,..\,i')} \mathsf{next}_{c_j}(S, \mathcal{L}) \neq \#$. Consequently, $\mathcal{X}$ is a λ-rainbow. We have $\min \mathcal{X} = \mathsf{prev}_{c_{i'}}(S, \mathcal{L})$. Further:

$$\max \mathcal{X} = \max(\{\mathsf{prev}_{c_\lambda}(S, \mathcal{L})\} \cup \{\mathsf{next}_{c_j}(S, \mathcal{L}) \,:\, j \in [1\,..\,i')\}) = \max\{\mathsf{next}_{c_j}(S, \mathcal{L}) \,:\, j \in [0\,..\,i)\}.$$

By Fact 2.1, $\mathsf{LCP}(\mathcal{X}) = \mathsf{LCP}(\min \mathcal{X}, \max \mathcal{X}) = \zeta$, as required.

$\zeta \geq \mathsf{LCP}(\mathcal{X}')$: Recall that $\mathcal{X}'$ was selected as a multiset that maximises $\min \mathcal{X}'$. Consider the multiset $\mathcal{Y}$ of minimal elements in $\mathcal{X}'$. We next argue that $\mathcal{Y}$ has a non-empty subset $\mathcal{Y}'$ such that for all $Y \in \mathcal{Y}'$, $Y = \mathsf{prev}_{\mathsf{col}(Y)}(S, \mathcal{L})$. Suppose to the contrary that this is not the case. Then, for each $Y \in \mathcal{Y}'$, let us replace Y in $\mathcal{X}'$ with $\mathsf{prev}_{\mathsf{col}(Y)}(S, \mathcal{L})$. This way, we would obtain a λ-rainbow $\mathcal{X}''$ containing S and satisfying $\min \mathcal{X}'' > \min \mathcal{X}'$. Moreover:

$$\mathsf{LCP}(\mathcal{X}'') = \mathsf{LCP}(\min \mathcal{X}'', \max \mathcal{X}'') = \mathsf{LCP}(\min \mathcal{X}'', \max \mathcal{X}') \geq \mathsf{LCP}(\min \mathcal{X}', \max \mathcal{X}') = \mathsf{LCP}(\mathcal{X}'),$$

by Fact 2.1 as $\min X' < \min \mathcal{X}'' \leq \max \mathcal{X}'$; a contradiction.

Now, consider an element of $\mathcal{Y}'$ with minimal colour, and suppose that this colour is c_i. Now, it suffices to note that:

$$\max(\mathcal{X}') \geq \max\{\mathsf{next}_{c_j}(S, \mathcal{L}) \ : \ j \in [0\,..\,i)\}. \tag{4}$$

Indeed, if $i > 1$, by the order of $c_1, \ldots, c_\lambda$, no string V in $\mathcal{L}$ such that $\mathsf{prev}_{c_i}(S, \mathcal{L}) \leq V \leq S$ has a colour in $\{c_1, \ldots, c_{i-1}\}$, and the smallest strings in $\mathcal{L}$ that are greater than S and have colours in $\{c_1, \ldots, c_{i-1}\}$ are exactly $\mathsf{next}_{c_j}(S, \mathcal{L})$ for $j \in [1\,..\,i)$. If $i = 1$, the right-hand side of Equation (4) is just S.

By Equation (4) and Fact 2.1:

$$\mathsf{LCP}(\mathcal{X}') = \mathsf{LCP}(\min \mathcal{X}', \max \mathcal{X}') \leq \mathsf{LCP}(\min \mathcal{X}', \max\{\mathsf{next}_{c_j}(S, \mathcal{L}) \ : \ j \in [0\,..\,i)\}) = \zeta,$$

as required.

Proof of part (a): From part (b), it follows that, for every string S:

$$\mathrm{maxLCP}(S, \mathcal{L}) = \mathsf{LCP}(\mathsf{next}(U, \mathcal{L}), U),$$

for some string U. More precisely, the string U is defined as the string $\mathsf{prev}_{c_i}(S, \mathcal{L})$ for i which yields the maximal value in Equation (3). Hence, the maximum LCP of a λ-rainbow $\mathcal{X} \subseteq \mathcal{L}$ does not exceed $\max_{U \in \mathcal{L}} \mathsf{LCP}(\mathsf{next}(U, \mathcal{L}), U)$, which is equal by symmetry to $\max_{U \in \mathcal{L}} \mathsf{LCP}(\mathsf{prev}(U, \mathcal{L}), U)$. The same value is clearly a lower bound for this maximum LCP. This concludes the proof. □

We often answer LCP queries between substrings of pairs of input strings $T^{(1)}, T^{(2)}, \ldots, T^{(\lambda)}$. To this end, we use the LCP data structure of Theorem 2.10 in a string $T^{(1)}\$T^{(2)}\$\cdots\$T^{(\lambda)}$ (where $\$ \notin \Sigma$). Whenever we perform a LCE query for substrings of two strings $T^{(x)}$ and $T^{(y)}$, we ensure that no delimiter \$ is crossed by capping the result—as the substrings are given in the form $T^{(z)}[i\,..\,j]$, we can infer the distances of i and j from the beginning and end of $T^{(z)}$ in constant time. A symmetric solution can be used for answering LCP queries on substrings of the reversals of the input strings. Let us note that we deliberately do not use unique delimiters when concatenating the input strings as this would blow up the alphabet size if σ is small.

5.2 Dynamic Predecessor Data Structures

For a set of integers A and integer x, the predecessor and successor of x in A are defined as $\max\{a \in A \ : \ a < x\}$ and $\min\{a \in A \ : \ a > x\}$, respectively. (We assume that $\max \emptyset = -\infty$ and $\min \emptyset = \infty$.) In a dynamic version of the problem, a set $A \subseteq [1\,..\,u]$ of size up to n is to be maintained under insertions and deletions so that predecessor and successor queries for A can be answered efficiently. The classic solution by van Emde Boas [92] achieves $O(\log \log u)$ time per operation using space $O(u)$. We use a slightly more space-efficient solution.

Lemma 5.2 ([75, 95]). *The dynamic predecessor/successor problem on an instance of size up to n over a universe $[1\,..\,u]$ can be solved in $O(\log \log u)$ time per operation using space $O(u/\log u + n)$.*

The dynamic predecessor/successor problem can also be solved in $O(n)$ space and $O(\log \log u)$ amortised expected time per operation using a y-fast trie [94] or in $O(\log^2 \log u / \log \log \log u)$

worst-case time per operation using an exponential search tree [7]. For simplicity, an AVL tree with $O(n)$ space and $O(\log u)$ time cost of operations can also be used. By π_u we denote the time of an operation on a predecessor/successor data structure of size $O(n)$ over universe $[1\,..\,u]$.

5.3 Short LCS

We assume that $\ell \leq \frac{1}{3}\log_\sigma n$.

The algorithm follows rather closely the proof of Lemma 3.1. Each of the input strings is split into fragments of length (up to) $2m$, where $m = \left\lfloor \frac{1}{3}\log_\sigma n \right\rfloor$. We compute all distinct substrings corresponding to such fragments for each of the strings in $O(n/m + \lambda + n^{2/3})$ total time using Radix Sort of pairs of the form: (an integer encoding a fragment, id number of the string it originates from). Each string $T^{(t)}$ is then replaced with a concatenation of its distinct fragments with unique delimiters. Thus, λ strings of total length $O(n^{2/3}m + \lambda)$ are obtained, and their LCS can be computed in $O(n^{2/3}m + \lambda)$ time [64]. The overall time complexity is $O(n/m + n^{2/3}m + \lambda) = O(n/\log_\sigma n)$.

5.4 Long LCS

Now, we assume that $\ell = \Omega(\frac{\log^{4\lambda} n}{\log^{\lambda} \sigma})$.

We consider a (λ, d)-cover (D, h) for $d \leq \ell$ and then take:

$-\mathcal{P}^{(t)} := \{((T^{(t)}[1\,..\,i))^R, T^{(t)}[i\,..\,|T^{(t)}|]) : i \in [1\,..\,|T^{(t)}|] \cap D\}$ for $t \in [1\,..\,\lambda]$;
$-\mathcal{F} := \{U : (U, V) \in \bigcup_{t=1}^{\lambda} \mathcal{P}^{(t)}$ or $(V, U) \in \bigcup_{t=1}^{\lambda} \mathcal{P}^{(t)}$ for some string $V\}$.

Thus, the LCS problem for λ strings is reduced to an instance of λ String Families LCP Problem for $\mathcal{P}^{(1)}, \ldots, \mathcal{P}^{(\lambda)}$ and $\mathcal{F}$. By Theorem 1.6, $|\mathcal{P}^{(t)}| = O(\lambda |T^{(t)}|/\sqrt[\lambda]{d})$, so $N = |\mathcal{F}| + \sum_{t=1}^{\lambda} |\mathcal{P}^{(t)}| = O(\lambda \cdot n/\sqrt[\lambda]{d})$. By Theorem 1.6, the sets $\mathcal{P}^{(1)}, \ldots, \mathcal{P}^{(\lambda)}$ can be constructed in $O(\lambda \cdot n/\sqrt[\lambda]{d})$ time. Construction of the compacted trie $\mathcal{T}(\mathcal{F})$ in $O(N \log N + n/\log_\sigma n)$ time follows the construction for two strings.

The solution is completed by the following lemma. In its proof, we adapt the technique of merging AVL trees from [54]. A main modification is that we do not need to use the very efficient merging algorithm of Brown and Tarjan [28] in which case we would have to ensure that it can handle a more complex type of queries; instead, we pay an extra $O(\log n)$ factor that stems from using a simple smaller-to-larger trick.

Lemma 5.3. *The λ String Families LCP Problem can be solved in $O(N\lambda\pi_N \log N)$ time and $O(N\lambda)$ space.*

Proof. For strings $U_1, \ldots, U_k$, by $\mathsf{LCPStr}(U_1, \ldots, U_k)$ we denote their longest common prefix. For each node w of $\mathcal{T}(\mathcal{F})$, we will compute a collection of pairs $(P_1^{(1)}, P_2^{(1)}) \in \mathcal{P}^{(1)}, \ldots, (P_1^{(\lambda)}, P_2^{(\lambda)}) \in \mathcal{P}^{(\lambda)}$ such that $\mathsf{val}(w)$ is a prefix of $\mathsf{LCPStr}(P_1^{(1)}, \ldots, P_1^{(\lambda)})$ (where $\mathsf{val}(w)$ is the string corresponding to node w) and $\mathsf{LCP}(P_2^{(1)}, \ldots, P_2^{(\lambda)})$ is maximal; the latter will be denoted as $\mathsf{result}(w)$ (if there is no such collection, we set $\mathsf{result}(w) = -\infty$). In the end, we will return the maximum of values $|\mathsf{val}(w)| + \mathsf{result}(w)$ over nodes w. Let us observe that the prefix $\mathsf{LCPStr}(P_1^{(1)}, \ldots, P_1^{(\lambda)})$ corresponds to an explicit node of $\mathcal{T}(\mathcal{F})$, so it is sufficient to consider explicit nodes w.

We will iterate through all explicit nodes of $\mathcal{T}(\mathcal{F})$ in a bottom-up manner. For each node w, we will compute the multiset:

$$\mathcal{L}(w) = \{V \,:\, (U, V) \in \bigcup_{t=1}^{\lambda} \mathcal{P}^{(t)},\ \mathsf{val}(w) \text{ is a prefix of } U\}.$$

Computing the Sets $\mathcal{L}(w)$. We order the second components of pairs across all the sets $\mathcal{P}^{(t)}$ lexicographically. This can be done in $O(N)$ time by a left-to-right traversal of $\mathcal{T}(\mathcal{F})$. For each

string pair (U, V) in a set $\mathcal{P}^{(t)}$, we assign to V an integer identifier $\mathsf{id}(V)$ that indicates the position of V in the sorted list of strings and colour $\mathsf{col}(V) = t$.

The strings in $\mathcal{L}(w)$ will be stored in λ dynamic predecessor data structures, one per string colour. The strings V in a predecessor data structure will be stored using their identifiers $\mathsf{id}(V)$. We will ensure that, at any time, there is an injection from the elements stored in the predecessor structures to the elements of the multiset obtained as the union of $\mathcal{P}^{(t)}$ for $t \in [1\,..\,\lambda]$. Hence, the total space required for the predecessor structures is $O(N\lambda)$.

The multisets $\mathcal{L}(w)$ can be initialised to empty in $O(N\lambda)$ time. For an explicit node w, we compute $\mathcal{L}(w)$ as $\bigcup_{j=1}^{r} \mathcal{L}(v_j)$ where $v_1, \ldots, v_r$ are all explicit children of w; we may then need to insert additional strings if w is a terminal node (that is, we insert strings V from pairs (U, V) from $\bigcup_{t=1}^{\lambda} \mathcal{P}^{(t)}$ such that $U = \mathsf{val}(w)$). The multisets $\mathcal{L}(v_j)$ are not kept; instead, as usual in applications of the smaller-to-larger trick, the union is computed by choosing the largest among the multisets, say $\mathcal{L}(v_{j'})$, and inserting to it all elements from $\mathcal{L}(v_j)$, for $j \neq j'$ (elements are moved from source predecessor data structure with colour c to a target predecessor data structure of the same colour), and potentially strings obtained for a terminal node. In total, every element will be moved between predecessor data structures at most $O(\log N)$ times, as the size of its multiset $\mathcal{L}$ at least doubles after a move, and the final multiset $\mathcal{L}$ for the root of $\mathcal{T}(\mathcal{F})$ has $O(N)$ elements. Thus, the total cost of insertions is $O(N\lambda + N\pi_N \log N)$.

Computing the Maximum LCP of a Rainbow. The multiset $\mathcal{L}(w)$ can be viewed as a multiset of strings with assigned colours. When the multiset $\mathcal{L}(w)$ has been computed, for each string V that was inserted to it (that is, $V \in \mathcal{L}(v_j)$ for some $j \neq j'$), we will compute the maximum LCP of a λ-rainbow $\mathcal{X} \subseteq \mathcal{L}(w)$ such that $V \in \mathcal{X}$. We compute $\mathsf{result}(w)$ as the maximum of these LCPs (or $-\infty$ if there is no such LCP) and the values $\mathsf{result}(v_j)$ over all explicit children v_j of w.

The maximum LCP of a λ-rainbow $\mathcal{X}$ containing a given $V \in \mathcal{L}(w)$ is computed using the formula from Lemma 5.1(b). Our representation of the multiset $\mathcal{L}(w)$ allows answering each query $\mathsf{prev}_c(S, \mathcal{L}(w))$ or $\mathsf{next}_c(S, \mathcal{L}(w))$ for $S \in \mathcal{L}(w)$ in $O(\pi_N)$ time, the time of an operation on a predecessor/successor data structure of linear size over $[1\,..\,N]$. The formula takes $O(\lambda\pi_N)$ time to compute the values prev and next, $O(\lambda \log\log \lambda)$ time to compute the permutation $c_1, \ldots, c_\lambda$ by sorting [63], and $O(\lambda)$ time for the remaining operations; in total, it works in $O(\lambda(\log\log \lambda + \pi_N)) = O(\lambda\pi_N)$ time as sorting reduces to dynamic predecessors and $\lambda \leq N$. The formula is used $O(N \log N)$ times, whenever a string is inserted into a multiset $\mathcal{L}$. This yields $O(N\lambda\pi_N \log N)$ time. □

Recall that $N = O(\lambda \cdot n/\sqrt[\lambda]{d})$. Overall, the time complexity of the solution to long LCS (Lemma 5.3) is $O(n\lambda^2\pi_n \log n/\sqrt[\lambda]{d} + n/\log_\sigma n)$, which is $O(n/\log_\sigma n)$ if $d = \Omega(\frac{\log^{4\lambda} n}{\log^{\lambda} \sigma})$ and $\lambda = O(\sqrt{\log n})$ (and $\pi_n = O(\log n)$). Clearly, the space is also bounded by $O(n/\log_\sigma n)$ in this case.

5.5 Medium-Length LCS

Finally, we consider the case that $\frac{1}{3}\log_\sigma n \leq \ell \leq 2^{O(\sqrt{\log n})}$. Then, all possible values of ℓ will have been considered, as $\log^{4\lambda} n = 2^{O(\sqrt{\log n})}$ for $\lambda = O(\sqrt{\log n}/\log\log n)$.

The subsets of positions A_j for $j \in \{I, II, III\}$ are defined as in Section 3.2 for a string:

$$T^{(1)}\$ \cdots T^{(\lambda-1)}\$T^{(\lambda)},$$

where $\$ \notin \Sigma$. We then denote:

$$A_j^{(t)} = \{a - sum_{t-1} \,:\, a \in A_j \cap (sum_{t-1}\,..\,sum_t)\},$$

for $j \in \{I, II, III\}$ and $sum_t = |T^{(1)}| + \cdots + |T^{(t)}| + t$. Note that given the sets A_j for $j \in \{I, II, III\}$ and the lengths of the strings $T^{(t)}$ for $t \in [1\,..\,\lambda]$, we can compute all sets $A_j^{(t)}$ in time $\mathcal{O}(\lambda + \sum_{j\in\{I,II,III\}} |A_j| \log\log n)$ using integer sorting [63]. This yields a time complexity of $\mathcal{O}(n \log\log n/\log_\sigma n)$.

Lemma 3.5 can be directly generalised as follows.

Lemma 5.4. *If an LCS of $T^{(1)}, \ldots, T^{(\lambda)}$ has length $\ell \geq 3\tau$, then there exist positions $i^{(t)} \in [1\,..\,|T^{(t)}|]$ for $t \in [1\,..\,\lambda]$, a shift $\delta \in [0\,..\,\ell)$ and $j \in \{I, II, III\}$ such that all strings $T^{(t)}[i^{(t)}\,..\,i^{(t)} + \ell)$ for $t \in [1\,..\,\lambda]$ are equal, $i^{(t)} + \delta \in A_j^{(t)}$ for all $t \in [1\,..\,\lambda]$, and:*

- *if $j = I$, then we can choose $\delta \in [0\,..\,\tau)$;*
- *if $j = II$, then $T^{(1)}[i^{(1)}\,..\,i^{(1)}+\ell)$ is contained in the τ-run from which $i^{(1)}+\delta \in A^{(1)}$ originates;*
- *if $j = III$, then $\delta \geq 3\tau-2$ and $T^{(1)}[i^{(1)}\,..\,i^{(1)}+\delta]$ is a suffix of the τ-run from which $i^{(1)}+\delta \in A^{(1)}$ originates.*

The proof of Lemma 5.4 follows the proof of Lemma 3.5; we choose the indices $i^{(1)}, \ldots, i^{(\lambda)}$ that minimise the sum $i^{(1)} + \cdots + i^{(\lambda)}$. Let us note that if $\ell \geq 3\tau$, the equal delimiters do not create any τ-runs in $T^{(1)}\$\cdots T^{(\lambda-1)}\$T^{(\lambda)}$ that would not be present in the strings $T^{(1)}, \ldots, T^{(\lambda)}$.

Let us now show how to generalise the solutions to the special cases of Two String Families LCP Problem to λ strings.

Lemma 5.5. *An instance of the λ String Families LCP Problem in which $\bigcup_{t=1}^{\lambda} \mathcal{P}^{(t)}$ is a prefix family can be solved in $\mathcal{O}(N\lambda \log\log N)$ time and $\mathcal{O}(N\lambda/\log N + N)$ space.*

Proof. We first show how Claim 3.7 from the proof of Lemma 3.6 can be adapted to the case of λ input strings.

We construct an array $\mathcal{R}$ that contains all string pairs from $\bigcup_{t=1}^{\lambda} \mathcal{P}^{(t)}$, sorted by their second components. For a string pair $e = (U, V) \in \mathcal{P}^{(t)}$, where $t \in [1\,..\,\lambda]$, we denote a list of strings:

$$\mathcal{L}_{|U|} = \{V' \,:\, (U', V') \in \mathcal{R},\ |U'| \geq |U|\}.$$

We assume that strings in $\mathcal{L}_{|U|}$ are sorted lexicographically and each string V' in $\mathcal{L}_{|U|}$ has a colour $c \in [1\,..\,\lambda]$ such that the original string pair (U', V') belonged to $\mathcal{P}^{(c)}$. This allows us to apply the techniques behind Lemma 5.1.

The following claim replaces Claim 3.7. We assume that $\mathrm{maxLCP}(V, \mathcal{L}_{|U|}) = -\infty$ (cf. Lemma 5.1) if there is no λ-rainbow $\mathcal{X} \subseteq \mathcal{L}_{|U|}$ such that $V \in \mathcal{X}$.

Claim 5.6. $\mathrm{maxMultiLCP}(\mathcal{P}^{(1)}, \ldots, \mathcal{P}^{(\lambda)}) = \max_{(U,V)\in\mathcal{R}}(|U| + \mathrm{maxLCP}(V, \mathcal{L}_{|U|}))$.

Proof. We have $\mathrm{maxMultiLCP}(\mathcal{P}^{(1)}, \ldots, \mathcal{P}^{(\lambda)}) \geq \max_{(U,V)\in\mathcal{R}}(|U| + \mathrm{maxLCP}(V, \mathcal{L}_{|U|}))$ since $\bigcup_{t=1}^{\lambda} \mathcal{P}^{(t)}$ is a prefix family. We proceed to show the other inequality.

For all $t \in [1\,..\,\lambda]$, let us pick $(P_1^{(t)}, P_2^{(t)}) \in \mathcal{P}^{(t)}$ such that:

$$\mathrm{maxMultiLCP}(\mathcal{P}^{(1)}, \ldots, \mathcal{P}^{(\lambda)}) = \mathsf{LCP}(P_1^{(1)}, \ldots, P_1^{(\lambda)}) + \mathsf{LCP}(P_2^{(1)}, \ldots, P_2^{(\lambda)}).$$

As in the proof of Claim 3.7, we choose an index $t' \in [1\,..\,\lambda]$ such that $\min\{|P_1^{(1)}|, \ldots, |P_1^{(\lambda)}|\} = |P_1^{(t')}|$. Let us denote $(U, V) = (P_1^{(t')}, P_2^{(t')})$. Then:

$$\mathsf{LCP}(P_1^{(1)}, \ldots, P_1^{(\lambda)}) = |P_1^{(t')}| = |U| \text{ and } \mathsf{LCP}(P_2^{(1)}, \ldots, P_2^{(\lambda)}) \leq \mathrm{maxLCP}(V, \mathcal{L}_{|U|}).$$

Hence, indeed $\mathrm{maxMultiLCP}(\mathcal{P}^{(1)}, \ldots, \mathcal{P}^{(\lambda)}) \leq \max_{(U,V)\in\mathcal{R}}(|U| + \mathrm{maxLCP}(V, \mathcal{L}_{|U|}))$. □

Algorithm 3: Application of LCPs Lists in the Case of λ Input Strings

$\ell_1 := \cdots := \ell_\lambda := r := 0$;
while not G.end_of_stream() **do**
  $\ell := \mathcal{L}$.read();
  for $t := 1$ **to** λ **do** $\ell_t := \min(\ell_t, \ell)$;
  $g := G$.read();
  $\ell_g := \beta$;
  $r := \max(r, \min(\ell_1, \ldots, \ell_\lambda))$;
return r;

Lemma 5.1(b) gives a formula for $\mathrm{maxLCP}(V, \mathcal{L}_{|U|})$ that can be evaluated efficiently if strings $\mathsf{prev}_c(V, \mathcal{L}_{|U|})$ (and $\mathsf{next}_c(V, \mathcal{L}_{|U|})$) for all $c \in [1\,..\,\lambda]$ are at hand and strings $\mathsf{prev}_c(V, \mathcal{L}_{|U|})$ can be sorted by c.

We will process all pairs in $\mathcal{R}$ in the order of non-decreasing lengths of the first components. As before, this order can be computed in $O(N)$ time using the compacted trie $\mathcal{T}(\mathcal{F})$. All elements with equal lengths of the first components are processed simultaneously.

We store integer sets $L^{(1)}, \ldots, L^{(\lambda)}$ such that $L^{(t)}$ contains an index $i \in [1\,..\,|\mathcal{R}|]$ if and only if $\mathcal{R}[i]$ originates from $\mathcal{P}^{(t)}$ and was not processed yet. Each set $L^{(t)}$ is stored in an instance of the dynamic predecessor/successor data structure of Lemma 5.2, for a total of $O(N\lambda/\log N + N)$ space.

Consider the processing of a string pair $\mathcal{R}[i] = (U, V)$. At the time of its processing, we have:

$$L^{(t)} = \{j \in [1\,..\,|\mathcal{R}|] \,:\, \mathcal{R}[j] = (U', V'),\ |U'| \geq |U|,\ \mathsf{col}(V') = t\},$$

for each $t \in [1\,..\,\lambda]$. Thus, to answer a $\mathsf{prev}_c(V, \mathcal{L}_{|U|})$ query, it suffices to compute the predecessor of i in $L^{(c)}$. If the predecessor is infinite, $\mathsf{prev}_c(V, \mathcal{L}_{|U|}) = \$$. If the predecessor equals $j < i$, then $\mathsf{prev}_c(V, \mathcal{L}_{|U|}) = V'$ where $\mathcal{R}[j] = (U', V')$. Values $\mathsf{next}_c(V, \mathcal{L}_{|U|})$ can be computed symmetrically using successor queries. Finally, all strings $\mathsf{prev}_c(V, \mathcal{L}_{|U|})$, for $c \in [1\,..\,\lambda]$, can be sorted by sorting the corresponding predecessor values.

Hence, computing $\mathrm{maxLCP}(V, \mathcal{L}_{|U|})$ requires $O(\lambda \log\log N)$ time for asking λ predecessor and successor queries (Lemma 5.2) and $O(\lambda \log\log \lambda)$ time for sorting strings $\mathsf{prev}_c(V, \mathcal{L}_{|U|})$, for $c \in [1\,..\,\lambda]$, represented as integers [63]. LCPs on second components of pairs are computed in $O(1)$ time using LCA queries on $\mathcal{T}(\mathcal{F})$. Overall, this gives $O(N\lambda \log\log N)$ time. □

We now generalise the solution to the second special case of λ String Families LCP Problem.

Lemma 5.7. *An instance of the λ String Families LCP Problem in which $\mathcal{P}^{(t)}$, for all $t \in [1\,..\,\lambda]$, are (α, β)-families can be solved in time $O(N(\alpha + \log N)(\log\beta + \sqrt{\log N})/\log N)$ and space $O(N + N\alpha/\log N)$ provided that $\log\beta = o(\log N)$ and $\lambda = o(\sqrt{\log N})$.*

Proof. To obtain Lemma 5.7, we use wavelet trees as in the proof of Lemma 1.5 in Section 4.

Lists of LCPs are computed using Algorithm 1. The sole difference is that a local variable used for storing the value from stream G now uses $\log\lambda$ bits instead of just one bit. The space grows to $s = O(\log\beta + \log\lambda)$ bits.

For applying LCPs list, we use Algorithm 3. Let $\mathcal{R}_v = (U_1, V_1), \ldots, (U_m, V_m)$ and $\mathcal{V}_v = V_1, \ldots, V_m$ be a list of strings with colours in $[1\,..\,\lambda]$ that correspond to their origins G_v. We next show that,

after the ith step of the while-loop, we have:

$$\ell_t = \begin{cases} \mathsf{LCP}(V_i, \mathsf{prev}_t(V_i, \mathcal{V}_v)) & \text{if } G[i] \neq t, \\ \beta & \text{otherwise,} \end{cases}$$
$$r = \max(\{0\} \cup \{\mathsf{LCP}(V_j, \mathsf{prev}(V_j, \mathcal{V}_v)) \,:\, j \in [1\,..\,i]\}).$$

After the first step of the loop ($\ell = 0$), we indeed have $\ell_t = 0$ for all $t \in [1\,..\,\lambda] \setminus \{g\}$ and $\ell_g = \beta$, where $g = G[1]$. Let us consider the $(i+1)$th step of the loop with $g = G[i+1]$. We have $\ell = \mathsf{LCP}(V_i, V_{i+1}) \leq \beta$, so, for each $t \in [1\,..\,\lambda] \setminus \{g\}$, the variable ℓ_t becomes:

$$\mathsf{LCP}(V_i, V_{i+1}) \text{ if } G[i] = t \text{ and } \min(\mathsf{LCP}(V_i, \mathsf{prev}_t(V_i, \mathcal{V}_v)), \mathsf{LCP}(V_i, V_{i+1})) \text{ otherwise.}$$

In either case, $\ell_t = \mathsf{LCP}(V_{i+1}, \mathsf{prev}_t(V_{i+1}, \mathcal{V}_v))$; if $G[i] \neq t$, we use Fact 2.1 and the fact that the strings in the list $\mathcal{V}_v$ are ordered lexicographically to argue for this. Then, ℓ_g is set to β, which is an obvious upper bound on $\mathsf{LCP}(V_{i+1}, \mathsf{prev}_g(V_{i+1}, \mathcal{V}_v)) = \mathsf{LCP}(V_{i+1}, V_{i+1})$. In the end, the value of r is maximised with:

$$\min\{\mathsf{LCP}(V_{i+1}, \mathsf{prev}_t(V_{i+1}, \mathcal{V}_v)) \,:\, t \in [1\,..\,\lambda] \setminus \{g\}\} = \mathsf{LCP}(V_{i+1}, \mathsf{prev}(V_{i+1}, \mathcal{V}_v)).$$

Hence, the value of r is as claimed. The correctness follows from Lemma 5.1(a).

Algorithm 3 works on $O(1)$ data streams, uses $s = O(\lambda \log \beta + \log \lambda) = o(\log n)$ (for $\lambda = o(\sqrt{\log n})$) bits of space, and performs a read or write among every $t = O(\lambda)$ instructions.

For a node v of the wavelet tree, the total size of the input and output streams in bits is proportional to the number of bits in streams $\mathcal{L}_v$ and G_v, i.e., $O(|B_v|(\log \beta + \log \lambda))$. By Proposition 2.4, for $\tau = \frac{1}{2} \log n$, after $o(N)$ preprocessing, each of the algorithms can be executed in $O(1 + |B_v|(\log \beta + \log \lambda)/\log N)$ time. Over all nodes, Algorithms 1 and 3 consume $O(Nh(\log \beta + \log \lambda)/\log N)$ time, where $h = O(\alpha + \log N)$. As in Lemma 1.5, the wavelet tree can be built in $O(Nh/\sqrt{\log N} + N)$ time using space $O(Nh/\log N + N)$. This yields Lemma 5.7. □

Finally, Lemma 5.4 leads to an $O(n \log\log n / \log_\sigma n)$-time reduction of computing LCS of λ strings if $\frac{1}{3}\log_\sigma n \leq \ell \leq 2^{O(\sqrt{\log n})}$ to several instances of λ String Families LCP Problem of total size $N = O(n/\tau)$, just as in the proof of Lemma 3.8. For each instance, either all sets $\mathcal{P}^{(t)}$ are $(\frac{1}{3}\log_\sigma n, 2^{O(\sqrt{\log n})})$-families (and then Lemma 5.7 can be used) or $\bigcup_{t=1}^{\lambda} \mathcal{P}^{(t)}$ is a prefix family (and then Lemma 5.5 is applied). The total time complexity is $O(n\lambda \log\log n \log\sigma/\log n + n \log\sigma/\sqrt{\log n})$, which is $O(n \log\sigma/\sqrt{\log n})$ for $\lambda = O(\sqrt{\log n}/\log\log n)$. The space complexity is $O(n\lambda \log\sigma/\log^2 n + n/\log_\sigma n)$ which is $O(n/\log_\sigma n)$ if $\lambda = O(\log n)$.

Together with the $O(n/\log_\sigma n)$-time algorithms of Sections 5.3 and 5.4, we obtain the following result, which extends Theorem 1.1.

Theorem 5.8. *Given $\lambda = O(\sqrt{\log n}/\log\log n)$ strings $T^{(1)}, \ldots, T^{(\lambda)}$ of total length n over an alphabet $[0\,..\,\sigma)$, their LCS can be computed in $O(n \log\sigma/\sqrt{\log n})$ time using $O(n \log\sigma/\log n)$ space.*

6 Faster k-LCS

In this section, we present our $O(n \log^{k-1/2} n)$-time algorithm for the k-LCS problem with $k = O(1)$, that underlies Theorem 1.2. For simplicity, we focus on computing the length of a k-LCS; an actual pair of strings forming a k-LCS can be recovered easily from our approach. If the length of an LCS of S and T is d, then the length of a k-LCS of S and T is in $[d\,..\,(k+1)d + k]$. Below, we show how to compute a k-LCS provided that it belongs to an interval $(\lceil \ell/2 \rceil\,..\,\ell]$ for a specified ℓ; to solve the k-LCS problem it suffices to call this subroutine $O(\log k) = O(1)$ times.

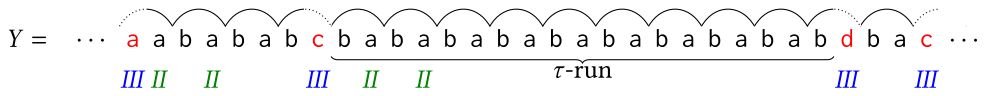


Fig. 7. A τ-run for $\tau = 6$ with period $p = 2$ and Lyndon root ab. For $k = 1$ and i being the first position of the τ-run, the set $\mathsf{Misper}_{k+1}(Y, i, i+p)$ contains the letters shown in red. Positions that are added to the sets A_{II} and A_{III} because of this τ-run are shown. Intuitively, if a length-ℓ substring U of Y starts within p next positions of a position in $\mathsf{LeftMisper}_{k+1}(Y, i, i+p)$, it has to contain a position from A_{II}, and otherwise, if U fits within the approximately periodic substring of Y, the substring U' shifted by p positions to the left contains the same (or smaller) set of misperiods.

Definition 6.1. For two strings $U, V \in \Sigma^m$, we define the *mismatch positions* $\mathsf{MP}(U, V) = \{i \in [1\,..\,m] : U[i] \neq V[i]\}$. Moreover, for $k \in \mathbb{Z}_{\geq 0}$, we write $U =_k V$ if $|\mathsf{MP}(U, V)| \leq k$.

Similarly to our solutions for long and medium-length LCS, we first distinguish anchors $A^S \subseteq [1\,..\,|S|]$ in S and $A^T \subseteq [1\,..\,|T|]$ in T, as summarised in the following lemma.

Lemma 6.2. *Consider an instance of the k-LCS problem for $k = O(1)$ and let $\ell \in [1\,..\,n]$. In $O(n)$ time, one can construct sets $A^S \subseteq [1\,..\,|S|]$ and $A^T \subseteq [1\,..\,|T|]$ of size $O(\frac{n}{\ell})$ satisfying the following condition: If a k-LCS of S and T has length $\ell' \in (\lceil \ell/2 \rceil\,..\,\ell]$, then there exist positions $i^S \in [1\,..\,|S|]$, $i^T \in [1\,..\,|T|]$ and a shift $\delta \in [0\,..\,\ell')$ such that $i^S + \delta \in A^S$, $i^T + \delta \in A^T$, and the Hamming distance between $S[i^S\,..\,i^S + \ell')$ and $T[i^T\,..\,i^T + \ell')$ is at most k.*

Proof. As in [40], we say that position a in a string X is a *misperiod* with respect to a substring $X[i\,..\,j)$ if $X[a] \neq X[b]$, where b is the unique position such that $b \in [i\,..\,j)$ and $(j-i) \mid (b-a)$; for example, $j - i$ is a period of X if and only if there are no misperiods with respect to $X[i\,..\,j)$. We define the set $\mathsf{LeftMisper}_k(X, i, j)$ as the set of k rightmost misperiods to the left of i and $\mathsf{RightMisper}_k(X, i, j)$ as the set of k leftmost misperiods to the right of j. Either set may have fewer than k elements if the corresponding misperiods do not exist. Further, let us define $\mathsf{Misper}_k(X, i, j) = \mathsf{LeftMisper}_k(X, i, j) \cup \mathsf{RightMisper}_k(X, i, j)$ and $\mathsf{Misper}(X, i, j) = \bigcup_{k=0}^{\infty} \mathsf{Misper}_k(X, i, j)$.

Similar to Lemma 3.5, we construct three subsets of positions in $Y = \#S\$T$, where $\#, \$ \notin \Sigma$. For $\tau = \lfloor \ell/(6(k+1)) \rfloor$, let A_I be a τ-synchronising set of Y. Let $Y[i\,..\,j]$ be a τ-run with period p and assume that the first occurrence of its Lyndon root is at a position q of Y. Then, for $Y[i\,..\,j]$, for each $x \in \mathsf{LeftMisper}_{k+1}(Y, i, i+p)$, we insert to A_{II} the two smallest positions in $[x+1\,..\,|Y|]$ that are equivalent to $q \pmod p$. Moreover, we insert to A_{III} the positions in $\mathsf{Misper}_{k+1}(Y, i, i+p)$. See Figure 7. Finally, we denote $A = A_I \cup A_{II} \cup A_{III}$, as well as $A^S = \{a - 1 : a \in A \cap [2\,..\,|S|+1]\}$ and $A^T = \{a - |S| - 2 : a \in A \cap [|S|+3\,..\,|Y|]\}$. The proof of the following claim resembles that of Lemma 3.5.

Claim 6.3. The sets A^S and A^T satisfy the condition stated in Lemma 6.2.

Proof. By the assumption of Lemma 6.2, a k-LCS of S and T has length $\ell' \in (\ell/2, \ell]$. Consequently, there exist $i^S \in [1\,..\,|S|]$ and $i^T \in [1\,..\,|T|]$ such that $U =_k V$, for $U = S[i^S\,..\,i^S + \ell')$ and $V = T[i^T\,..\,i^T + \ell')$. Let us choose any such pair (i^S, i^T) minimising the sum $i^S + i^T$. We shall prove that there exists $\delta \in [0\,..\,\ell')$ such that $i^S + \delta \in A^S$ and $i^T + \delta \in A^T$. In each of the three main cases, we actually show that $i^S + \delta \in A^S_j$ and $i^T + \delta \in A^T_j$ for $\delta \in [0\,..\,\ell')$ and for some $j \in \{I, II, III\}$, where $A^S_j = \{a - 1 : a \in A_j \cap [2\,..\,|S|+1]\}$ and $A^T_j = \{a - |S| - 2 : a \in A_j \cap [|S|+3\,..\,|Y|]\}$ (recall that $Y = \#S\$T$).

Recall that $\tau = \lfloor \ell/(6(k+1)) \rfloor$. By the pigeonhole principle, there exists a shift $s \in [0\,..\,\ell' - 3\tau + 2]$ such that:

$$W := S[i^S + s\,..\,i^S + s + 3\tau - 2] = T[i^T + s\,..\,i^T + s + 3\tau - 2].$$

First, assume that $\mathrm{per}(W) > \frac{1}{3}\tau$. By the definition of a τ-synchronising set, in this case there exist some elements $a^S \in A_I^S \cap [i^S + s \,..\, i^S + s + \tau)$ and $a^T \in A_I^T \cap [i^T + s \,..\, i^T + s + \tau)$. Let us choose the smallest such elements. By Lemma 2.7, we have $a^S - i^S = a^T - i^T$.

From now on we consider the case that $p = \mathrm{per}(W) \leq \frac{1}{3}\tau$. Thus, each of the two distinguished fragments of S and T that match W belongs to a τ-run with period p.

In the case where $\mathsf{Misper}_{k+1}(U, 1+s, 1+s+p) \cap \mathsf{Misper}_{k+1}(V, 1+s, 1+s+p) \neq \emptyset$, there exist $a^S \in A_{III}^S$ and $a^T \in A_{III}^T$ that satisfy the desired condition. This is because, for any string Z and its run $Z[i \,..\, j]$ with period p and $i' \in [i \,..\, j-p+1]$, $\mathsf{Misper}_{k+1}(Z, i, i+p) = \mathsf{Misper}_{k+1}(Z, i', i'+p)$.

In order to handle the complementary case, we rely on the following claim. We recall its short proof for completeness.

Claim 6.4 ([40, Lemma 13]). Assume that $U =_k V$ and that $U[i \,..\, j) = V[i \,..\, j)$. Let:

$$I = \mathsf{Misper}_{k+1}(U, i, j) \text{ and } I' = \mathsf{Misper}_{k+1}(V, i, j).$$

If $I \cap I' = \emptyset$, then $\mathsf{MP}(U, V) = I \cup I'$, $I = \mathsf{Misper}(U, i, j)$ and $I' = \mathsf{Misper}(V, i, j)$.

Proof. Let $J = \mathsf{Misper}(U, i, j)$ and $J' = \mathsf{Misper}(V, i, j)$. We first observe that $I \cup I' \subseteq \mathsf{MP}(U, V)$ since $I \cap I' = \emptyset$. Then, $U =_k V$ implies that $|\mathsf{MP}(U, V)| \leq k$ and hence $|I| \leq k$ and $|I'| \leq k$, which in turn implies that $I = J$ and $I' = J'$. The observation that $\mathsf{MP}(U, V) \subseteq J \cup J'$ concludes the proof. □

Towards a contradiction, let us suppose that there are no $a^S \in A_{II}^S$ and $a^T \in A_{II}^T$ satisfying the desired condition. By Claim 6.4, we have:

$$|\mathsf{LeftMisper}_{k+1}(U, 1+s, 1+s+p)|, |\mathsf{LeftMisper}_{k+1}(V, 1+s, 1+s+p)| \leq k.$$

Therefore, at least one of the following holds:

$$i^S \in \mathsf{LeftMisper}_{k+1}(\#S, 1+i^S+s, 1+i^S+s+p) \text{ or}$$
$$i^T \in \mathsf{LeftMisper}_{k+1}(\$T, 1+i^T+s, 1+i^T+s+p);$$

otherwise, $S[i^S - 1] = T[i^T - 1]$ and $(i^S - 1, i^T - 1)$ has a smaller sum and could be used instead of (i^S, i^T); a contradiction. Let us assume that the first of the two above conditions holds; the other case is symmetric. Then, $[i^S \,..\, i^S + p)$ contains an element of A_{II}^S, say a^S. If $A_{II}^T \cap [i^T \,..\, i^T + p) = \emptyset$, then this implies that:

$$[i^T - p + 1 \,..\, i^T] \cap \mathsf{LeftMisper}_{k+1}(\$T, 1+i^T+s, 1+i^T+s+p) = \emptyset.$$

In particular, we have $i^T > p$. Now, let us consider U and $V' = T[i^T - p \,..\, i^T + \ell' - p)$. By Claim 6.4, we have that $|\mathsf{Misper}(U, 1+s, 1+s+p)| + |\mathsf{Misper}(V, 1+s, 1+s+p)| \leq k$. Further, we have $|\mathsf{Misper}(V', 1+s+p, 1+s+2p)| \leq |\mathsf{Misper}(V, 1+s, 1+s+p)|$. Thus, $|\mathsf{MP}(U, V')| \leq |\mathsf{Misper}(U, 1+s, 1+s+p)| + |\mathsf{Misper}(V', 1+s, 1+s+p)| \leq k$. This contradicts our assumption that $i^S + i^T$ was minimum possible.

Consequently, there exists an element $a^T \in A_{II}^T \cap [i^T \,..\, i^T + p)$. By modular arithmetics, we have $\delta := a^S - i^S = a^T - i^T$. Then δ certainly belongs to $[0 \,..\, \ell')$, which concludes the proof. □

It remains to show that the sets A^S and A^T can be constructed efficiently. A τ-synchronising set can be computed in $O(n)$ time by Theorem 2.5 and all the τ-runs, together with the position of the first occurrence of their Lyndon root, can be computed in $O(n)$ time [13]. After an $O(n)$-time preprocessing, for every τ-run, we can compute the set of the $k+1$ misperiods of its period to either side in $O(1)$ time; see [40, Claim 18]. □

The next step in our solutions to long LCS and medium-length LCS was to construct an instance of the Two String Families LCP Problem. To adapt this approach, we generalise the notions of

LCP and maxPairLCP so that they allow for mismatches. By $\mathsf{LCP}_k(U,V)$, for $k \in \mathbb{Z}_{\geq 0}$, we denote the maximum length ℓ such that $U[1\,..\,\ell]$ and $V[1\,..\,\ell]$ are at Hamming distance at most k.

Definition 6.5. Given two sets $\mathcal{U},\mathcal{V} \subseteq \Sigma^* \times \Sigma^*$ and two integers $k_1, k_2 \in \mathbb{Z}_{\geq 0}$, we define $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V}) = \max\{\mathsf{LCP}_{k_1}(U_1,V_1) + \mathsf{LCP}_{k_2}(U_2,V_2) : (U_1,U_2) \in \mathcal{U}, (V_1,V_2) \in \mathcal{V}\}$.

Note that $\mathrm{maxPairLCP}(\mathcal{U},\mathcal{V}) = \mathrm{maxPairLCP}_{0,0}(\mathcal{U},\mathcal{V})$.
By Lemma 6.2, if a k-LCS of S and T has length $\ell' \in (\ell/2, \ell]$, then:

$$\ell' = \max_{k'=0}^{k} \mathrm{maxPairLCP}_{k',k-k'}(\mathcal{U},\mathcal{V}), \text{ for } \mathcal{U} = \{((S[a-\ell\,..\,a))^R, S[a\,..\,a+\ell)) : a \in A^S\},$$
$$\mathcal{V} = \{((T[a-\ell\,..\,a))^R, T[a\,..\,a+\ell)) : a \in A^T\}.$$

Here, k' bounds the number of mismatches between $S[i^S\,..\,i^S+\delta)$ and $T[i^T\,..\,i^T+\delta)$, whereas $k-k'$ bounds the number of mismatches between $S[i^S+\delta\,..\,i^S+\ell')$ and $T[i^T+\delta\,..\,i^T+\ell')$. The following theorem, whose full proof is given in Section 7, allows for efficiently computing the values $\mathrm{maxPairLCP}_{k',k-k'}(\mathcal{U},\mathcal{V})$.

Theorem 6.6. *Consider two (ℓ,ℓ)-families $\mathcal{U},\mathcal{V}$ of total size N consisting of pairs of substrings of a given length-n text over the alphabet $[0\,..\,n)$. For any non-negative integers $k_1, k_2 = O(1)$, the value $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V})$ can be computed*:

- *in $O(n + N\log^{k_1+k_2+1} N)$ time and $O(n+N)$ space if $\ell > \log^{3/2} N$,*
- *in $O(n + N\ell\log^{k_1+k_2-1/2} N)$ time and $O(n + N\ell/\log N)$ space if $\log N < \ell \leq \log^{3/2} N$ and*
- *in $O(n + N\ell^{k_1+k_2}\sqrt{\log N})$ time and $O(n+N)$ space if $\ell \leq \log N$.*

Here we just give an outline of the proof. We reduce the computation of $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V})$ into multiple computations of $\mathrm{maxPairLCP}(\mathcal{U}',\mathcal{V}')$ across a family $\mathbf{P}$ of pairs $(\mathcal{U}',\mathcal{V}')$ with $\mathcal{U}',\mathcal{V}' \subseteq \Sigma^* \times \Sigma^*$. Each pair $(U_1',U_2') \in \mathcal{U}'$ is associated to a pair $(U_1,U_2) \in \mathcal{U}$, with the string U_i' represented as a pointer to the *source* U_i and up to k_i substitutions needed to transform U_i to U_i'. Similarly, each pair $(V_1',V_2') \in \mathcal{V}'$ consists of *modified strings* with sources $(V_1,V_2) \in \mathcal{V}$. In order to guarantee $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V}) = \max_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}} \mathrm{maxPairLCP}(\mathcal{U}',\mathcal{V}')$, we require $\mathsf{LCP}(U_i',V_i') \leq \mathsf{LCP}_{k_i}(U_i,V_i)$ for every $(U_1',U_2') \in \mathcal{U}'$ and $(V_1',V_2') \in \mathcal{V}'$ with $(\mathcal{U}',\mathcal{V}') \in \mathbf{P}$ and that, for every $(U_1,U_2) \in \mathcal{U}$ and $(V_1,V_2) \in \mathcal{V}$, there exists $(\mathcal{U}',\mathcal{V}') \in \mathbf{P}$ with $(U_1',U_2') \in \mathcal{U}'$ and $(V_1',V_2') \in \mathcal{V}'$, with sources (U_1,U_2) and (V_1,V_2), respectively, such that $\mathsf{LCP}(U_i',V_i') = \mathsf{LCP}_{k_i}(U_i,V_i)$. Our construction is based on a technique of [89] which gives an analogous family for two subsets of Σ^* (rather than $\Sigma^* \times \Sigma^*$) and a single threshold. We apply the approach of [89] to $\mathcal{U}_i = \{U_i : (U_1,U_2) \in \mathcal{U}\}$ and $\mathcal{V}_i = \{V_i : (V_1,V_2) \in \mathcal{V}\}$ with threshold k_i, and then combine the two resulting families to derive $\mathbf{P}$.

Strengthening the arguments of [89], we show that each string $F_i \in \mathcal{U}_i \cup \mathcal{V}_i$ is the source of $O(1)$ modified strings $F_i' \in \mathcal{U}_i' \cup \mathcal{V}_i'$ for any single $(\mathcal{U}_i',\mathcal{V}_i') \in \mathbf{P}_i$ and $O(\min(\ell,\log N)^{k_i})$ modified strings across all $(\mathcal{U}_i',\mathcal{V}_i') \in \mathbf{P}_i$. This allows bounding the size of individual sets $(\mathcal{U}',\mathcal{V}') \in \mathbf{P}$ by $O(N)$ and the overall size by $O(N\min(\ell,\log N)^{k_1+k_2})$. In order to efficiently build the compacted tries required at the input of the Two String Families LCP Problem, the modified strings $F_i' \in \mathcal{U}_i' \cup \mathcal{V}_i'$ are sorted lexicographically, and the two derived linear orders (for $i \in \{1,2\}$) are maintained along with every pair $(\mathcal{U}',\mathcal{V}') \in \mathbf{P}$. Overall, the family $\mathbf{P}$ is constructed in $O(n + N\min(\ell,\log N)^{k_1+k_2})$ time and $O(n+N)$ space.

The resulting instances of the Two String Families LCP Problem are solved using Lemma 1.3 if $\ell > \log^{3/2} N$ or Lemma 1.5 otherwise; note that $\mathcal{U}',\mathcal{V}'$ are (ℓ,ℓ)-families.

Recall that the algorithm of Theorem 6.6 is called $k+1 = O(1)$ times, always with $N = |A^S| + |A^T| = O(n/\ell)$. Overall, the value $\max_{k'=0}^{k} \mathrm{maxPairLCP}_{k',k-k'}(\mathcal{U},\mathcal{V})$ is therefore computed in $O(n \log^{k-1/2} n)$ time and $O(n)$ space in each of the following cases:

- in $O(n + \frac{n}{\ell}\log^{k+1} N) = O(n \log^{k-1/2} n)$ time and $O(n + \frac{n}{\ell}) = O(n)$ space if $\ell > \log^{3/2} N$;
- in $O(n + \frac{n}{\ell}\ell \log^{k-1/2} N) = O(n \log^{k-1/2} n)$ time and $O(n + \frac{n}{\ell}\ell/\log N) = O(n)$ space if $\log N < \ell \le \log^{3/2} N$;
- in $O(n + \frac{n}{\ell}\ell^k\sqrt{\log N}) = O(n \log^{k-1/2} n)$ time and $O(n + \frac{n}{\ell}) = O(n)$ space if $\ell \le \log N$.

Accounting for $O(n)$ time and space to determine the length d of an LCS between S and T, and the $O(\log k)$ values ℓ that need to be tested so that the intervals $(\lceil \ell/2 \rceil .. \ell]$ cover $[d .. (k+1)d + k]$, this implies Theorem 1.2.

We also obtain the following corollary that improves [34] for $k = O(1)$. (We replace $\sqrt{\ell}$ by ℓ.)

Corollary 6.7. *Given two strings S and T of total length n and a constant integer $k > 0$, the k-LCS problem can be solved in $O(n + \frac{n}{\ell}\log^{k+1} n)$ time using $O(n)$ space, where ℓ is the length of the k-LCS.*

Proof. The three cases yield the following time complexities: in $O(n + \frac{n}{\ell}\log^{k+1} n)$ time if $\ell > \log^{3/2} N$, in $O(n \log^{k-1/2} n) = O(\frac{n}{\ell}\log^{k+1} n)$ time if $\log N < \ell \le \log^{3/2} N$, and in $O(n \log^{k-1/2} n) = O(\frac{n}{\ell}\log^{k+1} n)$ time if $\ell \le \log N$. This gives an $O(n + \frac{n}{\ell}\log^{k+1} n)$-time solution for any ℓ. The space complexity is still $O(n)$. □

7 Computing $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V})$—Proof of Theorem 6.6

For two strings $U, U' \in \Sigma^m$, we define the *mismatch positions* $\mathsf{MP}(U,U') = \{i \in [1..n] : U[i] \neq U'[i]\}$ and the *mismatch information* $\mathsf{MI}(U,U') = \{(i,U'[i]) : i \in \mathsf{MP}(U,U')\}$. Observe that U and $\mathsf{MI}(U,U')$ uniquely determine U'. This motivates the following definition.

Definition 7.1. Given $U \in \Sigma^m$ and $\Delta \subseteq [1..m] \times \Sigma$, we denote by U^Δ the unique string U' such that $\mathsf{MI}(U,U') = \Delta$. If there is no such string U', then U^Δ is undefined. We say that U^Δ, represented using a pointer to U and the set Δ, is a *modified string* with *source* U.

Example 7.2. Let $U = \mathsf{ababbab}$ and $\Delta = \{(2,\mathsf{a}),(3,\mathsf{b})\}$. Then $U^\Delta = U' = \mathsf{aabbbab}$.

Definition 7.3 (See [89]). Given strings $U, V \in \Sigma^*$ and an integer $k \in \mathbb{Z}_{\ge 0}$, we say that two modified strings (U^Δ, V^∇) form a $(U,V)_k$*-maxpair* if the following holds for every i:

- if $i \in [1..\mathsf{LCP}_k(U,V)]$ and $U[i] \neq V[i]$, then $U^\Delta[i] = V^\nabla[i]$;
- otherwise, $U^\Delta[i] = U[i]$ (assuming $i \in [1..|U|]$) and $V^\nabla[i] = V[i]$ (assuming $i \in [1..|V|]$).

Example 7.4. Let $U = \mathsf{ababbabb}$, $V = \mathsf{aacbaaab}$ and $k = 3$. Further let $\Delta = \{(2,\mathsf{a}),(3,\mathsf{b})\}$ and $\nabla = \{(3,\mathsf{b}),(5,\mathsf{b}))\}$. We have $U^\Delta = \mathsf{aabbbabb}$ and $U^\nabla = \mathsf{aabbbaab}$. Then $\mathsf{LCP}_k(U,V) = 6$ and $(U^\Delta, V^\nabla) = (\mathsf{aabbba}\underline{\mathsf{ab}}, \mathsf{aabbba}\underline{\mathsf{bb}})$ form a $(U,V)_3$-maxpair. Additionally, for $\Delta' = \{(3,\mathsf{c})\}$ and $\nabla' = \{(2,\mathsf{b}),(5,\mathsf{b}))\}$, $(U^{\Delta'}, V^{\nabla'}) = (\mathsf{abcbba}\underline{\mathsf{ab}}, \mathsf{abcbba}\underline{\mathsf{bb}})$ form a $(U,V)_3$-maxpair.

The following simple fact characterises this notion. Fact 7.5. *Let U^Δ, V^∇ be modified strings with sources $U, V \in \Sigma^*$ and let $k \in \mathbb{Z}_{\ge 0}$.*

(a) *If (U^Δ, V^∇) is a $(U,V)_k$-maxpair, then $|\Delta \cup \nabla| \le k$ and $\mathsf{LCP}(U^\Delta, V^\nabla) \ge \mathsf{LCP}_k(U,V)$.*
(b) *If $|\Delta \cup \nabla| \le k$, then $\mathsf{LCP}_k(U,V) \ge \mathsf{LCP}(U^\Delta, V^\nabla)$.*

Proof.

(a) Let $d = \mathsf{LCP}_k(U, V)$ and $M = \mathsf{MP}(U[1\,..\,d], V[1\,..\,d])$. By Definition 7.3, we have $U^{\Delta}[i] = V^{\nabla}[i]$ for $i \in [1\,..\,d]$. Consequently, $\mathsf{LCP}(U^{\Delta}, V^{\nabla}) \geq d$ and, if $(i, a) \in \Delta$ and $(i, b) \in \nabla$ holds for some $i \in [1\,..\,d]$, then $a = b$. Furthermore, Definition 7.3 yields $\Delta, \nabla \subseteq M \times \Sigma$, and hence $|\Delta \cup \nabla| \leq |M| \leq k$.

(b) Let $d' = \mathsf{LCP}(U^{\Delta}, V^{\nabla})$ and $M' = \mathsf{MP}(U[1\,..\,d'], V[1\,..\,d'])$. For every $i \in M'$, we have $U[i] \neq V[i]$ yet $U^{\Delta}[i] = V^{\nabla}[i]$. This implies $U[i] \neq U^{\Delta}[i]$ or $V[i] \neq V^{\nabla}[i]$, i.e., $(i, U^{\Delta}[i]) = (i, V^{\nabla}[i]) \in \Delta \cup \nabla$. Consequently, $|M'| \leq |\Delta \cup \nabla| \leq k$, which means that $\mathsf{LCP}_k(U, V) \geq d'$ holds as claimed. □

In particular, if (U^{Δ}, V^{∇}) is a $(U, V)_k$-maxpair, then by Fact 7.5, $\mathsf{LCP}(U^{\Delta}, V^{\nabla}) = \mathsf{LCP}_k(U, V)$.

Definition 7.6. Consider a set of strings $\mathcal{F} \subseteq \Sigma^*$ and an integer $k \in \mathbb{Z}_{\geq 0}$. A *$k$-complete family* for $\mathcal{F}$ is a family $\mathbf{F}$ of sets of modified strings of the form F^{Δ} for $F \in \mathcal{F}$ and $|\Delta| \leq k$ such that, for every $U, V \in \mathcal{F}$, there exists a set $\mathcal{F}' \in \mathbf{F}$ and modified strings $U^{\Delta}, V^{\nabla} \in \mathcal{F}'$ forming a $(U, V)_k$-maxpair.

Example 7.7. Let us consider the following string family:

$$\mathcal{F}_1 = \{A_1 = \mathsf{aaaab},\ A_2 = \mathsf{abbac},\ A_3 = \ \mathsf{acbab},\ A_4 = \mathsf{bcaa},\ A_5 = \mathsf{bcbac}\}.$$

Family $\mathbf{F}_1 = \{\mathcal{F}'_{1,1}, \mathcal{F}'_{1,2}\}$ is a 1-complete family for $\mathcal{F}_1$, where:

$$\mathcal{F}'_{1,1} = \{\begin{aligned} A_1 &= \mathsf{aaaab},\\ A_3^{\{(2,\mathsf{a})\}} &= \mathsf{aabab},\\ A_2^{\{(2,\mathsf{a})\}} &= \mathsf{aabac},\\ A_2 &= \mathsf{abbac},\\ A_4^{\{(1,\mathsf{a})\}} &= \mathsf{acaa},\\ A_3 &= \mathsf{acbab},\\ A_5^{\{(1,\mathsf{a})\}} &= \mathsf{acbac},\\ A_4 &= \mathsf{bcaa},\\ A_5 &= \mathsf{bcbac}\,\}. \end{aligned} \qquad \mathcal{F}'_{1,2} = \{\begin{aligned} A_4 &= \mathsf{bcaa},\\ A_5^{\{(3,\mathsf{a})\}} &= \mathsf{bcaac},\\ A_5 &= \mathsf{bcbac}\,\}. \end{aligned}$$

For example, both strings in a $(A_2, A_3)_1$-maxpair ($A_2^{\{(2,\mathsf{a})\}} = \mathsf{aabac}, A_3^{\{(2,\mathsf{a})\}} = \mathsf{aabab}$) belong to $\mathcal{F}'_{1,1}$, and both strings in a $(A_4, A_5)_1$-maxpair ($A_4 = \mathsf{bcaa}, A_5^{\{(3,\mathsf{a})\}} = \mathsf{bca\ \ ac}$) belong to $\mathcal{F}'_{1,2}$.
Figure C1 in Appendix C shows how the family $\mathbf{F}_1$ was created.

Example 7.8. Let $\mathcal{X}$ and $\mathcal{Y}$ be the sets of all suffixes of strings S and T, respectively, and $\mathbf{F}$ be a k-complete family for $\mathcal{X} \cup \mathcal{Y}$. Then:

$$k\text{-LCS}(S, T) = \max_{\mathcal{F}' \in \mathbf{F}} \{\mathsf{LCP}(X^{\Delta}, Y^{\nabla}) \,:\, X \in \mathcal{X},\ Y \in \mathcal{Y},\ |\Delta \cup \nabla| \leq k,\ X^{\Delta}, Y^{\nabla} \in \mathcal{F}'\}.$$

Our construction of a k-complete family follows the approach of Thankachan et al. [89] (which, in turn, is based on the ideas behind k-errata trees of Cole et al. [46]) with minor modifications. For completeness, we provide a full proof of the following proposition in Appendix C.

Proposition 7.9. *Let $\mathcal{F} \subseteq \Sigma^{\leq \ell}$ and $k \in \mathbb{Z}_{\geq 0}$ with $k = O(1)$. There exists a k-complete family $\mathbf{F}$ for $\mathcal{F}$ such that, for each $F \in \mathcal{F}$:*

— *Every individual set* $\mathcal{F}' \in \mathbf{F}$ *contains* $O(1)$ *modified strings with source* F.
— *In total, the sets* $\mathcal{F}' \in \mathbf{F}$ *contain* $O(\min(\ell, \log|\mathcal{F}|)^k)$ *modified strings with source* F.

Moreover, if $\mathcal{F}$ *consists of substrings of a given length-*n *text over the alphabet* $[0\,..\,n)$*, then the family* $\mathbf{F}$ *can be constructed in* $O(n + |\mathcal{F}|)$ *space and* $O(n + |\mathcal{F}| \min(\ell, \log|\mathcal{F}|)^k)$ *time with sets* $\mathcal{F}' \in \mathbf{F}$ *generated one by one and modified strings within each set* $\mathcal{F}' \in \mathbf{F}$ *sorted lexicographically.*

Intuitively, in the approach of Thankachan et al. [89], a k-LCS was computed as the maximum LCP_k of any two suffixes originating from different strings S, T. Hence, using the k-complete family shown in Example 7.8 was sufficient. However, in our approach, a k-LCS is anchored at some pair of synchronised positions. This motivates the following generalised notion aimed to account for the parts of a k-LCS on both sides of the anchor.

Definition 7.10. Consider a set $\mathcal{G} \subseteq \Sigma^* \times \Sigma^*$ of string pairs and integers $k_1, k_2 \in \mathbb{Z}_{\geq 0}$. A (k_1, k_2)-*bicomplete family* for $\mathcal{G}$ is a family $\mathbf{G}$ of sets $\mathcal{G}'$ of modified string pairs of the form $(F_1^{\Delta_1}, F_2^{\Delta_2})$ for $(F_1, F_2) \in \mathcal{G}$, $|\Delta_1| \leq k_1$ and $|\Delta_2| \leq k_2$, such that, for every $(U_1, U_2), (V_1, V_2) \in \mathcal{G}$, there exists a set $\mathcal{G}' \in \mathbf{G}$ with $(U_1^{\Delta_1}, U_2^{\Delta_2}), (V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{G}'$ such that $(U_1^{\Delta_1}, V_1^{\nabla_1})$ is a $(U_1, V_1)_{k_1}$-maxpair and $(U_2^{\Delta_2}, V_2^{\nabla_2})$ is a $(U_2, V_2)_{k_2}$-maxpair.

A concrete example of a bicomplete family (Example 7.13) is presented after an efficient construction of such a family (Lemma 7.12).

Example 7.11. Let $\mathcal{U}$ and $\mathcal{V}$ be the sets of string pairs as considered in Theorem 6.6 and $\mathbf{G}$ be a (k_1, k_2)-bicomplete family for $\mathcal{U} \cup \mathcal{V}$. Then:

$$\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V}) = \max_{\mathcal{G}' \in \mathbf{G}} \{\mathsf{LCP}(U_1^{\Delta_1}, V_1^{\nabla_1}) + \mathsf{LCP}(U_2^{\Delta_2}, V_2^{\nabla_2}) :$$
$$(U_1, U_2) \in \mathcal{U}, (V_1, V_2) \in \mathcal{V}, |\Delta_1 \cup \nabla_1| \leq k_1, |\Delta_2 \cup \nabla_2| \leq k_2, (U_1^{\Delta_1}, U_2^{\Delta_2}), (V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{G}'\}.$$

Complete families can be used to efficiently construct a bicomplete family as shown in the following lemma.

Lemma 7.12. *Let* $\mathcal{G}$ *be an* (ℓ, ℓ)*-family and* $k_1, k_2 \in \mathbb{Z}_{\geq 0}$ *with* $k_1, k_2 = O(1)$*. There exists a* (k_1, k_2)*-bicomplete family* $\mathbf{G}$ *for* $\mathcal{G}$ *such that, for each* $(F_1, F_2) \in \mathcal{G}$:

— *Every individual set* $\mathcal{G}' \in \mathbf{G}$ *contains* $O(1)$ *pairs of the form* $(F_1^{\Delta_1}, F_2^{\Delta_2})$.
— *In total, the sets* $\mathcal{G}' \in \mathbf{G}$ *contain* $O(\min(\ell, \log|\mathcal{G}|)^{k_1+k_2})$ *pairs of the form* $(F_1^{\Delta_1}, F_2^{\Delta_2})$.

Proof. Let $\mathcal{F}_1 = \{F_1 : (F_1, F_2) \in \mathcal{G}\}$ and $\mathcal{F}_2 = \{F_2 : (F_1, F_2) \in \mathcal{G}\}$. Moreover, for $i \in \{1, 2\}$, let $\mathbf{F}_i$ be the k_i-complete family for $\mathcal{F}_i$ obtained using Proposition 7.9. The family $\mathbf{G}$ is defined as follows:

$$\mathbf{G} = \{\{(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{F}_1' \times \mathcal{F}_2' : (F_1, F_2) \in \mathcal{G}\} : (\mathcal{F}_1', \mathcal{F}_2') \in \mathbf{F}_1 \times \mathbf{F}_2\} \setminus \{\emptyset\}.$$

Clearly, each set $\mathcal{G}' \in \mathbf{G}$ consists of modified string pairs of the form $(F_1^{\Delta_1}, F_2^{\Delta_2})$ with $(F_1, F_2) \in \mathcal{G}$, $|\Delta_1| \leq k_1$ and $|\Delta_2| \leq k_2$. Let us fix pairs $(U_1, U_2), (V_1, V_2) \in \mathcal{G}$. By Definition 7.6, for $i \in \{1, 2\}$, there exists a set $\mathcal{F}_i' \in \mathbf{F}_i$ and modified strings $U_i^{\Delta_i}, V_i^{\nabla_i} \in \mathcal{F}_i'$ that form a $(U_i, V_i)_{k_i}$-maxpair. The family $\mathbf{G}$ contains a set $\mathcal{G}' = \{(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{F}_1' \times \mathcal{F}_2' : (F_1, F_2) \in \mathcal{G}\}$ and this set $\mathcal{G}'$ contains both $(U_1^{\Delta_1}, U_2^{\Delta_2})$ and $(V_1^{\nabla_1}, V_2^{\nabla_2})$. Thus, $\mathbf{G}$ is a (k_1, k_2)-bicomplete family for $\mathcal{G}$.

Now, let us fix a pair $(F_1, F_2) \in \mathcal{G}$ in order to bound the number of modified string pairs of the form $(F_1^{\Delta_1}, F_2^{\Delta_2})$ contained in the sets $\mathcal{G}' \in \mathbf{G}$. If $(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{G}'$, where $\mathcal{G}'$ is constructed for $(\mathcal{F}_1', \mathcal{F}_2') \in \mathbf{F}_1 \times \mathbf{F}_2$, then $F_i^{\Delta_i} \in \mathcal{F}_i'$. By Proposition 7.9, for $i \in \{1, 2\}$, the set $\mathcal{F}_i' \in \mathbf{F}_i$ contains $O(1)$ modified strings with the source F_i. Consequently, the set $\mathcal{G}' \in \mathbf{G}$ contains $O(1)$ modified

string pairs of the form $(F_1^{\Delta_1}, F_2^{\Delta_2})$. Moreover, Proposition 7.9 implies that, for $i \in \{1,2\}$, the sets $\mathcal{F}_i' \in \mathbf{F}_i$ in total contain $O(\min(\ell, \log|\mathcal{F}_i|)^{k_i}) = O(\min(\ell, \log|\mathcal{G}|)^{k_i})$ modified strings of the form $F_i^{\Delta_i}$. Thus, the sets $\mathcal{G}' \in \mathbf{G}$ in total contain $O(\min(\ell, \log|\mathcal{G}|)^{k_1+k_2})$ modified string pairs of the form $(F_1^{\Delta_1}, F_2^{\Delta_2})$. □

Example 7.13. Let us consider the following string family:

$$\mathcal{F}_2 = \{B_3 = \mathsf{aacba},\ B_2 = \mathsf{abcba},\ B_1 = \ \mathsf{babab},\ B_4 = \mathsf{babc},\ B_5 = \mathsf{bbcba}\}.$$

Family $\mathbf{F}_2 = \{\mathcal{F}'_{2,1}, \mathcal{F}'_{2,2}\}$ is a 1-complete family for $\mathcal{F}_2$, where:

$$\mathcal{F}'_{2,1} = \{\begin{aligned} B_3 &= \mathsf{aacba},\\ B_2 &= \mathsf{abcba},\\ B_4^{\{(4,\mathsf{a})\}} &= \mathsf{baba},\\ B_1 &= \mathsf{babab},\\ B_4 &= \mathsf{babc},\\ B_3^{\{(1,\mathsf{b})\}} &= \mathsf{bacba},\\ B_5^{\{(2,\mathsf{a})\}} &= \mathsf{bacba},\\ B_5 &= \mathsf{bbcba},\\ B_2^{\{(1,\mathsf{b})\}} &= \mathsf{bbcba}\}. \end{aligned} \qquad \mathcal{F}'_{2,2} = \{\begin{aligned} B_3 &= \mathsf{aacba},\\ B_2 &= \mathsf{abcba},\\ B_3^{\{(2,\mathsf{b})\}} &= \mathsf{abcba}\}. \end{aligned}$$

Now let us consider the following $(5,5)$-family, where $\mathcal{F}_1 = \{A_1, A_2, A_3, A_4, A_5\}$ was considered in Example 7.7:

$$\mathcal{G} = \{(A_1 = \mathsf{aaaab}, B_1 = \mathsf{babab}),\ (A_2 = \ \mathsf{abbac}, B_2 = \mathsf{abcba}),\ (A_3 = \mathsf{acbab}, B_3 = \mathsf{aacba}),\ (A_4 = \mathsf{bcab}, B_4 = \mathsf{babc}),\ (A_5 = \mathsf{bcbac}, B_5 = \mathsf{bbcba})\}.$$

Family $\mathbf{G} = \{\mathcal{G}'_{1,1}, \mathcal{G}'_{1,2}, \mathcal{G}'_{2,1}\}$ constructed as in Lemma 7.12 using $\mathbf{F}_1$ and $\mathbf{F}_2$ is a $(1,1)$-bicomplete family for $\mathcal{G}$, where:

$$\begin{aligned}
\mathcal{G}'_{1,1} = \{&(A_1 = \mathsf{aaaab}, B_1 = \mathsf{babab})\} \cup\\
&\{A_2 = \mathsf{abbac}, A_2^{\{(2,\mathsf{a})\}} = \mathsf{a\ \ abac}\} \times \{B_2 = \mathsf{abcba}, B_2^{\{(1,\mathsf{b})\}} = \mathsf{bbcba}\} \cup\\
&\{A_3 = \mathsf{acbab}, A_3^{\{(2,\mathsf{a})\}} = \mathsf{a\ \ abab}\} \times \{B_3 = \mathsf{aacba}, B_3^{\{(1,\mathsf{b})\}} = \mathsf{bacba}\} \cup\\
&\{A_4 = \mathsf{bcaa}, A_4^{\{(1,\mathsf{a})\}} = \mathsf{acaa}\} \times \{B_4 = \mathsf{babc}, B_4^{\{(4,\mathsf{a})\}} = \mathsf{baba}\} \cup\\
&\{A_5 = \mathsf{bcbac}, A_5^{\{(1,\mathsf{a})\}} = \mathsf{acbac}\} \times \{B_5 = \mathsf{bbcba}, B_5^{\{(2,\mathsf{a})\}} = \mathsf{bacba}\},\\
\mathcal{G}'_{1,2} = \{&A_2 = \mathsf{abbac}, A_2^{\{(2,\mathsf{a})\}} = \mathsf{a\ \ abac}\} \times \{B_2 = \mathsf{abcba}\} \cup\\
&\{A_3 = \mathsf{acbab}, A_3^{\{(2,\mathsf{a})\}} = \mathsf{a\ \ abab}\} \times \{B_3 = \mathsf{aacba}, B_3^{\{(2,\mathsf{b})\}} = \mathsf{abcba}\},\\
\mathcal{G}'_{2,1} = \{&A_4 = \mathsf{bcaa}\} \times \{B_4 = \mathsf{babc}, B_4^{\{(4,\mathsf{a})\}} = \mathsf{baba}\} \cup\\
&\{A_5 = \mathsf{bcbac}, A_5^{\{(3,\mathsf{a})\}} = \mathsf{bc\ \ aac}\} \times \{B_5 = \mathsf{bbcba}, B_5^{\{(2,\mathsf{a})\}} = \mathsf{bacba}\}.
\end{aligned}$$

Recall that in this example, $\mathbf{F}_i = \{\mathcal{F}'_{i,1}, \mathcal{F}'_{i,2}\}$ for $i \in \{1,2\}$. For each $i, j \in \{1,2\}$, we consider all pairs of modified strings in $\mathcal{F}'_{1,i} \times \mathcal{F}'_{2,j}$ and if a pair originates from a pair in $\mathcal{G}$, it is added to $\mathcal{G}'_{i,j}$. We do not create the set $\mathcal{G}'_{2,2}$ as it would be empty.

Let us check on two examples that $\mathbf{G}$ satisfies the conditions of Definition 7.10.

For $(A_2 = \mathsf{abbac}, B_2 = \mathsf{abcba})$, $(A_3 = \mathsf{acbab}, B_3 = \mathsf{aacba}) \in \mathcal{G}$:

– $(A_2^{\{(2,\mathsf{a})\}} = \mathsf{aabac}, A_3^{\{(2,\mathsf{a})\}} = \mathsf{aabab})$ is a $(A_2, A_3)_1$-maxpair with $\mathsf{LCP}(A_2^{\{(2,\mathsf{a})\}}, A_3^{\{(2,\mathsf{a})\}}) = 4$,
– $(B_2 = \mathsf{abcba}, B_3^{\{(2,\mathsf{b})\}} = \mathsf{ab\ \ cba})$ is a $(B_2, B_3)_1$-maxpair with $\mathsf{LCP}(B_2, B_3^{\{(2,\mathsf{b})\}}) = 5$ and
– $(A_2^{\{(2,\mathsf{a})\}}, B_2), (A_3^{\{(2,\mathsf{a})\}}, B_3^{\{(2,\mathsf{b})\}}) \in \mathcal{G}'_{1,2}$.

Let us note that $A_2^{\{(2,\mathsf{a})\}}, A_3^{\{(2,\mathsf{a})\}}$ contain modifications on only one position and the same applies to $B_2, B_3^{\{(2,\mathsf{b})\}}$. Let $\mathcal{U} = \{(A_1, B_1), (A_2, B_2), (A_5, B_5)\}$ and $\mathcal{V} = \{(A_3, B_3), (A_4, B_4)\}$. One can check that:

$$\mathrm{maxPairLCP}_{1,1}(\mathcal{U}, \mathcal{V}) = \mathsf{LCP}(A_2^{\{(2,\mathsf{a})\}}, A_3^{\{(2,\mathsf{a})\}}) + \mathsf{LCP}(B_2, B_3^{\{(2,\mathsf{b})\}}) = 9.$$

Let us note that even though the set $\mathcal{G}'_{1,1}$ contains pairs of modified strings with sources A_2 and B_2 and pairs with sources A_3 and B_3, no such pair of pairs yields a $(B_2, B_3)_1$-maxpair.

For $(A_3 = \mathsf{acbab}, B_3 = \mathsf{aacba})$, $(A_5 = \mathsf{bcbac}, B_5 = \mathsf{bbcba}) \in \mathcal{G}$:

– $(A_3 = \mathsf{acbab}, A_5^{\{(1,\mathsf{a})\}} = \mathsf{acbac})$ is a $(A_3, A_5)_1$-maxpair with $\mathsf{LCP}(A_3, A_5^{\{(1,\mathsf{a})\}}) = 4$,
– $(B_3^{\{(1,\mathsf{b})\}} = \mathsf{bacba}, B_5 = \mathsf{\ \ bbcba})$ is a $(B_3, B_5)_1$-maxpair with $\mathsf{LCP}(B_3^{\{(1,\mathsf{b})\}}, B_5) = 1$ and
– $(A_3, B_3^{\{(1,\mathsf{b})\}}), (A_5^{\{(1,\mathsf{a})\}}, B_5) \in \mathcal{G}'_{1,1}$.

Let us note that $(A_5^{\{(1,\mathsf{a})\}}, B_5^{\{(2,\mathsf{a})\}} = \mathsf{bacba})$ also belongs to $\mathcal{G}'_{1,1}$ and $\mathsf{LCP}(B_3^{\{(1,\mathsf{b})\}}, B_5^{\{(2,\mathsf{a})\}}) = 5$. However, the two modified strings contain modifications at *two different positions*, so they do not form a $(B_3, B_5)_1$-maxpair. In particular, they could not be used when computing $\mathrm{maxPairLCP}_{1,1}(\mathcal{U}, \mathcal{V})$ according to the formula from Example 7.11.

The construction of a bicomplete family from Lemma 7.12 requires processing the complete families in batches to ensure that only linear space is used.

Lemma 7.14. *Let $\mathcal{G}$ be an (ℓ, ℓ)-family and $k_1, k_2 \in \mathbb{Z}_{\geq 0}$ with $k_1, k_2 = O(1)$. If $\mathcal{G}$ consists of pairs of substrings of a given length-n text over the alphabet $[0\,..\,n)$, then a (k_1, k_2)-bicomplete family $\mathbf{G}$ for $\mathcal{G}$ that satisfies the conditions of* Lemma 7.12 *can be constructed in $O(n + |\mathcal{G}|)$ space and $O(n + |\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1+k_2})$ time with sets $\mathcal{G}' \in \mathbf{G}$ generated one by one and each set $\mathcal{G}'$ sorted in two ways: according to the lexicographic order of the modified strings on the first and the second coordinate, respectively.*

Proof. We will show how to construct the bicomplete family from the proof of Lemma 7.12. We first generate the family $\mathbf{F}_1$ and batch it into subfamilies $\mathbf{F}'_1 \subseteq \mathbf{F}_1$ using the following procedure applied on top of the algorithm of Proposition 7.9 after it outputs a set $\mathcal{F}'_1 \in \mathbf{F}_1$. For each modified string $F_1^{\Delta_1} \in \mathcal{F}'_1$, we augment each pair of the form $(F_1, F_2) \in \mathcal{G}$ with a triple consisting of $F_1^{\Delta_1}$, the lexicographic rank of $F_1^{\Delta_1}$ within $\mathcal{F}'_1$, and the index of $\mathcal{F}'_1$ within $\mathbf{F}'_1$. After processing a set $\mathcal{F}'_1 \in \mathbf{F}_1$ in this manner, we resume the algorithm of Proposition 7.9 provided that the total number of triples stored is smaller than $n + |\mathcal{G}|$. Otherwise (and when the algorithm of Proposition 7.9 terminates), we declare the construction of the current batch $\mathbf{F}'_1 \subseteq \mathbf{F}_1$ complete.

For each batch $\mathbf{F}'_1 \subseteq \mathbf{F}_1$, we generate the family $\mathbf{F}_2$ and batch it into subfamilies $\mathbf{F}'_2 \subseteq \mathbf{F}_2$ using the following procedure applied on top of the algorithm of Proposition 7.9 after it outputs a set $\mathcal{F}'_2 \in \mathbf{F}_2$. For each modified string $F_2^{\Delta_2} \in \mathcal{F}'_2$, we retrieve all pairs of the form $(F_1, F_2) \in \mathcal{G}$ and iterate over the triples stored at (F_1, F_2). For each such triple, consisting of a modified string $F_1^{\Delta_1}$, the lexicographic rank of $F_1^{\Delta_1}$ within $\mathcal{F}'_1 \in \mathbf{F}'_1$, and the index of $\mathcal{F}'_1$ within $\mathbf{F}'_1$, we generate the corresponding triple for $F_2^{\Delta_2}$ and combine the two triples into a 6-tuple. After processing a set $\mathcal{F}'_2 \in \mathbf{F}_2$ in this manner, we resume the algorithm of Proposition 7.9 provided that the total number of 6-tuples stored is

smaller than $n + |\mathcal{G}|$. Otherwise (and when the algorithm of Proposition 7.9 terminates), we declare the construction of the current batch $\mathbf{F}'_2 \subseteq \mathbf{F}_2$ complete.

For each pair of batches $\mathbf{F}'_1, \mathbf{F}'_2$, we group the 6-tuples by the index of $\mathcal{F}'_1$ within $\mathbf{F}'_1$ and the index of $\mathcal{F}'_2$ within $\mathbf{F}'_2$, and we sort the 6-tuples in each group in two ways: by the lexicographic rank of $F_1^{\Delta_1}$ within $\mathcal{F}'_1$ and by the lexicographic rank of $F_2^{\Delta_2}$ within $\mathcal{F}'_2$. The keys used for sorting and grouping are integers bounded by $n^{O(1)}$, so we implement this step using radix sort. Finally, for each group, we create a set $\mathcal{G}'$ by preserving only the modified strings $(F_1^{\Delta_1}, F_2^{\Delta_2})$ out of each 6-tuple. We output $\mathcal{G}'$ along with the two linear orders: according to $F_1^{\Delta_1}$ and $F_2^{\Delta_2}$. It is easy to see that this yields $\mathcal{G}' = \{(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{F}'_1 \times \mathcal{F}'_2 : (F_1, F_2) \in \mathcal{G}\}$. Consequently, for the two batches $\mathbf{F}'_1, \mathbf{F}'_2$, the algorithm produces:

$$\mathbf{G}' = \{\{(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{F}'_1 \times \mathcal{F}'_2 : (F_1, F_2) \in \mathcal{G}\} : (\mathcal{F}'_1, \mathcal{F}'_2) \in \mathbf{F}'_1 \times \mathbf{F}'_2\} \setminus \{\emptyset\}.$$

Since, for $i \in \{1, 2\}$, each set $\mathcal{F}'_i \in \mathbf{F}_i$ belongs to exactly one batch $\mathbf{F}'_i$, across all pairs of batches we obtain the family $\mathbf{G}$ defined above.

We conclude with the complexity analysis. The generators of $\mathbf{F}_1$ and $\mathbf{F}_2$ use $O(n + |\mathcal{F}_1|) = O(n + |\mathcal{G}|)$ and $O(n + |\mathcal{F}_2|) = O(n + |\mathcal{G}|)$ space, respectively. Each set $\mathcal{F}'_1 \in \mathbf{F}_1$ contains $O(1)$ modified strings with the same source, so the number of triples generated for $\mathcal{F}'_1$ is $O(|\mathcal{G}|)$, so the number of triples generated for each batch $\mathbf{F}'_1$ is $O(n + |\mathcal{G}|)$. Similarly, each set $\mathcal{F}'_2 \in \mathbf{F}_2$ contains $O(1)$ modified strings with the same source, so the number of 6-tuples generated for each batch pair $\mathbf{F}'_1, \mathbf{F}'_2$ is $O(n + |\mathcal{G}|)$. The triples are removed after processing each batch $\mathbf{F}'_1$ and the 6-tuples are removed after processing each pair of batches $\mathbf{F}'_1, \mathbf{F}'_2$, so the space complexity of the entire algorithm is $O(n + |\mathcal{G}|)$.

As for the running time, note that the generator of $\mathbf{F}_1$ takes $O(n + |\mathcal{F}_1| \min(\ell, \log |\mathcal{F}_1|)^{k_1}) = O(n + |\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1})$ time. In the post-processing of the sets $\mathcal{F}'_1 \in \mathbf{F}_1$, by Proposition 7.9, for each $(F_1, F_2) \in \mathcal{G}$ we generate $O(\min(\ell, \log |\mathcal{G}|)^{k_1})$ triples. In total, this gives $O(|\mathcal{G}| \min(\ell, \log |\mathcal{F}_1|)^{k_1}) = O(|\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1})$ triples in $O(1)$ time per triple. The number of batches $\mathbf{F}'_1$ is therefore $O(1 + \frac{|\mathcal{G}|}{n+|\mathcal{G}|} \min(\ell, \log |\mathcal{G}|)^{k_1})$. For each such batch, we run the generator of $\mathbf{F}_2$, which takes $O(n + |\mathcal{F}_2| \min(\ell, \log |\mathcal{F}_2|)^{k_2}) = O(n + |\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_2})$ time. Across all batches $\mathbf{F}'_1$, this sums up to:

$$O\left(\left(1 + \frac{|\mathcal{G}|}{n+|\mathcal{G}|} \min(\ell, \log |\mathcal{G}|)^{k_1}\right)(n + |\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_2})\right) = O(n + |\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1+k_2}).$$

In the post-processing of the sets $\mathcal{F}'_2 \in \mathbf{F}_2$, we generate $\sum_{\mathcal{G}' \in \mathbf{G}} |\mathcal{G}'| = O(|\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1+k_2})$ 6-tuples, in $O(1)$ time per tuple. The number of batch pairs $\mathbf{F}'_1, \mathbf{F}'_2$ is therefore:

$$O\left(1 + \frac{|\mathcal{G}|}{n+|\mathcal{G}|} \min(\ell, \log |\mathcal{G}|)^{k_1} + \frac{|\mathcal{G}|}{n+|\mathcal{G}|} \min(\ell, \log |\mathcal{G}|)^{k_1+k_2}\right) = O\left(1 + \frac{|\mathcal{G}|}{n+|\mathcal{G}|} \min(\ell, \log |\mathcal{G}|)^{k_1+k_2}\right).$$

Each batch pair is processed in $O(n + |\mathcal{G}|)$ time, so this yields $O(n + |\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1+k_2})$ time in total. □

Next we will use bicomplete families to reduce the computation of $\text{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V})$ for (ℓ, ℓ)-families of substrings of a given text to a number of computations of $\text{maxPairLCP}(\mathcal{U}', \mathcal{V}')$ for (ℓ, ℓ)-families of modified substrings. Similarly to [89], we need to group the elements of a bicomplete family for $\mathcal{U} \cup \mathcal{V}$ by subsets of modifications so that, if two modified strings have a common modification, it is counted as one mismatch between them. Intuitively, we primarily want to avoid checking the conditions $|\Delta_i \cup \nabla_i| \le k_i$ for $i \in \{1, 2\}$ in the formula in Example 7.11.

Proposition 7.15. *Let $\mathcal{U}, \mathcal{V}$ be (ℓ, ℓ)-families and $k_1, k_2 \in \mathbb{Z}_{\ge 0}$ with $k_1, k_2 = O(1)$. There exists a family $\mathbf{P}$ such that:*

(1) *each element of* $\mathbf{P}$ *is a pair of sets* $(\mathcal{U}', \mathcal{V}')$*, where the elements of* $\mathcal{U}'$ *are of the form* $(U_1^{\Delta_1}, U_2^{\Delta_2})$ *for* $(U_1, U_2) \in \mathcal{U}$ *with* $|\Delta_1| \leq k_1$ *and* $|\Delta_2| \leq k_2$*, whereas the elements of* $\mathcal{V}'$ *are of the form* $(V_1^{\nabla_1}, V_2^{\nabla_2})$ *for* $(V_1, V_2) \in \mathcal{V}$ *with* $|\nabla_1| \leq k_1$ *and* $|\nabla_2| \leq k_2$;
(2) $\max_{(\mathcal{U}', \mathcal{V}') \in \mathbf{P}}(|\mathcal{U}'| + |\mathcal{V}'|) = O(|\mathcal{U}| + |\mathcal{V}|)$;
(3) $\sum_{(\mathcal{U}', \mathcal{V}') \in \mathbf{P}}(|\mathcal{U}'| + |\mathcal{V}'|) = O((|\mathcal{U}| + |\mathcal{V}|) \min(\ell, \log(|\mathcal{U}| + |\mathcal{V}|))^{k_1+k_2})$;
(4) $\max_{(\mathcal{U}', \mathcal{V}') \in \mathbf{P}} \text{maxPairLCP}(\mathcal{U}', \mathcal{V}') = \text{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V})$.

Proof. Let $\mathcal{G} = \mathcal{U} \cup \mathcal{V}$ and let $\mathbf{G}$ be the (k_1, k_2)-bicomplete family for $\mathcal{G}$ constructed as in Lemma 7.12. Given $\mathcal{G}' \in \mathbf{G}$ and $\delta_1, \delta_2 \subseteq \mathbb{Z}_{\geq 0} \times \Sigma$, we define a subset of $\mathcal{G}'$:

$$\mathcal{G}'_{\delta_1,\delta_2} = \{(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{G}' : \delta_1 \subseteq \Delta_1, \delta_2 \subseteq \Delta_2\}.$$

For all non-empty sets $\mathcal{G}'_{\delta_1,\delta_2}$ with $\mathcal{G}' \in \mathbf{G}$, and all integers $d_1 \in [0\,..\,k_1]$ and $d_2 \in [0\,..\,k_2]$, we insert to $\mathbf{P}$ a pair $(\mathcal{U}', \mathcal{V}')$, where:

$$\mathcal{U}' = \{(U_1^{\Delta_1}, U_2^{\Delta_2}) \in \mathcal{G}'_{\delta_1,\delta_2} : (U_1, U_2) \in \mathcal{U},\ |\Delta_1| \leq d_1,\ |\Delta_2| \leq d_2\},$$
$$\mathcal{V}' = \{(V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{G}'_{\delta_1,\delta_2} : (V_1, V_2) \in \mathcal{V},\ |\nabla_1| \leq k_1 + |\delta_1| - d_1,\ |\nabla_2| \leq k_2 + |\delta_2| - d_2\}.$$

Clearly, the elements of $\mathcal{U}'$ and $\mathcal{V}'$ satisfy requirement (1). Moreover, $|\mathcal{U}'| + |\mathcal{V}'| \leq 2|\mathcal{G}'| = O(|\mathcal{G}|) = O(|\mathcal{U}| + |\mathcal{V}|)$, so requirement (2) is fulfilled. Furthermore, each pair $(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{G}'$ belongs to $2^{|\Delta_1|+|\Delta_2|} \leq 2^{k_1+k_2} = O(1)$ sets $\mathcal{G}'_{\delta_1,\delta_2}$ and thus to $O(k_1k_2) = O(1)$ sets $\mathcal{U}'$ or $\mathcal{V}'$ created for $\mathcal{G}'$. Consequently, due to $\sum_{\mathcal{G}' \in \mathbf{G}} |\mathcal{G}'| = O(|\mathcal{G}| \min(\ell, \log|\mathcal{G}|)^{k_1+k_2})$, we have $\sum_{(\mathcal{U}', \mathcal{V}') \in \mathbf{P}}(|\mathcal{U}'| + |\mathcal{V}'|) = O(|\mathcal{G}| \min(\ell, \log|\mathcal{G}|)^{k_1+k_2}) = O((|\mathcal{U}| + |\mathcal{V}|) \min(\ell, \log(|\mathcal{U}| + |\mathcal{V}|))^{k_1+k_2})$, which yields requirement (3).

Our next goal is to prove $\text{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V}) = \max_{(\mathcal{U}', \mathcal{V}') \in \mathbf{P}} \text{maxPairLCP}(\mathcal{U}', \mathcal{V}')$ (requirement (4)).

Example 7.16. As in Example 7.4, consider $U = \mathsf{ababbabb}$, $V = \mathsf{aacbaaab}$, $k = 3$, $\Delta = \{(2, \mathsf{a}), (3, \mathsf{b})\}$ and $\nabla = \{(3, \mathsf{b}), (5, \mathsf{b})\}$. Recall that (U^Δ, V^∇) form a $(U, V)_3$-maxpair. We have $\delta = \Delta \cap \nabla = \{(3, \mathsf{b})\}$. U^Δ has $d = 2$ modifications, and V^∇ has $k + |\delta| - d = 2$ modifications. Note that, if we instead had $\nabla = \{(3, \mathsf{b}), (5, \mathsf{b}), (7, \mathsf{b})\}$, then $\mathsf{LCP}(U^\Delta, V^\nabla)$ would be greater than $\mathsf{LCP}_3(U, V)$, and we would thus not want to consider the pair (U^Δ, V^∇).

Let us fix $(\mathcal{U}', \mathcal{V}') \in \mathbf{P}$ generated for $\mathcal{G}'$, δ_1, δ_2, d_1 and d_2. For every $(U_1^{\Delta_1}, U_2^{\Delta_2}) \in \mathcal{U}'$ and $(V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{V}'$, we have $|\Delta_i \cup \nabla_i| = |\Delta_i| + |\nabla_i| - |\Delta_i \cap \nabla_i| \leq d_i + (k_i + |\delta_i| - d_i) - |\delta_i| \leq k_i$ for $i \in \{1, 2\}$ and thus, by Fact 7.5(b), $\mathsf{LCP}(U_i^{\Delta_i}, V_i^{\nabla_i}) \leq \mathsf{LCP}_{k_i}(U_i, V_i)$. Consequently, we have $\mathsf{LCP}(U_1^{\Delta_1}, V_1^{\nabla_1}) + \mathsf{LCP}(U_2^{\Delta_2}, V_2^{\nabla_2}) \leq \mathsf{LCP}_{k_1}(U_1, V_1) + \mathsf{LCP}_{k_2}(U_2, V_2)$, and thus $\text{maxPairLCP}(\mathcal{U}', \mathcal{V}') \leq \text{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V})$.

As for the converse inequality, suppose that $(U_1, U_2) \in \mathcal{U}$ and $(V_1, V_2) \in \mathcal{V}$ satisfy:

$$\text{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V}) = \mathsf{LCP}_{k_1}(U_1, V_1) + \mathsf{LCP}_{k_2}(U_2, V_2).$$

By the definition of a bicomplete family (Definition 7.10), there exist $\mathcal{G}' \in \mathcal{G}$ and pairs of modified strings $(U_1^{\Delta_1}, U_2^{\Delta_2}), (V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{G}'$ such that $(U_i^{\Delta_i}, V_i^{\nabla_i})$ is a $(U_i, V_i)_{k_i}$-maxpair for $i \in \{1, 2\}$. By Fact 7.5, this yields $\mathsf{LCP}_{k_i}(U_i, V_i) = \mathsf{LCP}(U_i^{\Delta_i}, V_i^{\nabla_i})$ and $|\Delta_i \cup \nabla_i| \leq k_i$. We define $\delta_i = \Delta_i \cap \nabla_i$ and $d_i = |\Delta_i|$ for $i \in \{1, 2\}$, and consider the pair $(\mathcal{U}', \mathcal{V}')$ constructed for $\mathcal{G}'$, δ_1, δ_2, d_1 and d_2. Note that

$|\Delta_i| \le d_i$ and $\delta_i \subseteq \Delta_i$, so $(U_1^{\Delta_1}, U_2^{\Delta_2}) \in \mathcal{U}'$. Moreover, $|\nabla_i| = |\Delta_i \cup \nabla_i| + |\delta_i| - |\Delta_i| \le k_i + |\delta_i| - d_i$ and $\delta_i \subseteq \nabla_i$, so $(V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{V}'$. Consequently, $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V}) \le \mathrm{maxPairLCP}(\mathcal{U}',\mathcal{V}')$.

□

Example 7.17. Let us consider the sets $\mathcal{U} = \{(A_1, B_1), (A_2, B_2), (A_5, B_5)\}$ and $\mathcal{V} = \{(A_3, B_3), (A_4, B_4)\}$ and the $(1,1)$-bicomplete family $\mathbf{G}$ of $\mathcal{U} \cup \mathcal{V}$ from Example 7.13. Let $\mathcal{G}'$ be the following set $\mathcal{G}'_{1,2}$ from that example:

$$\mathcal{G}' = \{A_2 = \mathsf{abbac}, A_2^{\{(2,\mathsf{a})\}} = \mathsf{a\ \ abac}\} \times \{B_2 = \mathsf{abcba}\} \cup$$
$$\{A_3 = \mathsf{acbab}, A_3^{\{(2,\mathsf{a})\}} = \mathsf{a\ \ abab}\} \times \{B_3 = \mathsf{aacba}, B_3^{\{(2,\mathsf{b})\}} = \mathsf{abcba}\}.$$

In the construction of Proposition 7.15, in particular, the following set is constructed:

$$\mathcal{G}'_{\{(2,\mathsf{a})\},\emptyset} = \{A_2^{\{(2,\mathsf{a})\}} = \mathsf{aabac}\} \times \{B_2 = \mathsf{abcba}\} \cup$$
$$\{A_3^{\{(2,\mathsf{a})\}} = \mathsf{aabab}\} \times \{B_3 = \mathsf{aacba}, B_3^{\{(2,\mathsf{b})\}} = \mathsf{abcba}\}.$$

For $\mathcal{G}'_{\{(2,\mathsf{a})\},\emptyset}$, $d_1 = 1$ and $d_2 = 0$ the following pair of sets $(\mathcal{U}', \mathcal{V}')$ is inserted to $\mathbf{P}$:

$$\mathcal{U}' = \{A_2^{\{(2,\mathsf{a})\}} = \mathsf{aabac}\} \times \{B_2 = \mathsf{abcba}\},$$
$$\mathcal{V}' = \{A_3^{\{(2,\mathsf{a})\}} = \mathsf{aabab}\} \times \{B_3 = \mathsf{aacba}, B_3^{\{(2,\mathsf{b})\}} = \mathsf{abcba}\}.$$

We have $\mathrm{maxPairLCP}_{1,1}(\mathcal{U},\mathcal{V}) = \mathrm{maxPairLCP}(\mathcal{U}',\mathcal{V}')$, as discussed in Example 7.13.

As another example, $\mathcal{G}'_{\emptyset,\emptyset} = \mathcal{G}'$. For this set, no $d_1, d_2 \in [0\,..\,1]$ produce a pair of sets $(\mathcal{U}', \mathcal{V}')$ such that $(A_2^{\{(2,\mathsf{a})\}}, B_2) \in \mathcal{U}'$, $(A_3^{\{(2,\mathsf{a})\}}, B_3^{\{(2,\mathsf{b})\}}) \in \mathcal{V}'$. Intuitively, this is because in $\mathcal{G}'_{\emptyset,\emptyset}$, the modifications $\{(2,\mathsf{a})\}$ in the first modified strings in the pairs are counted twice.

Lemma 7.18. *Let $\mathcal{U}, \mathcal{V}$ be (ℓ,ℓ)-families and $k_1, k_2 \in \mathbb{Z}_{\ge 0}$ with $k_1, k_2 = O(1)$. Moreover, if $\mathcal{U}$ and $\mathcal{V}$ consist of pairs of substrings of a given length-n text over the alphabet $[0\,..\,n)$, then the family $\mathbf{P}$ from Proposition 7.15 can be constructed in $O(n + (|\mathcal{U}| + |\mathcal{V}|)\min(\ell, \log(|\mathcal{U}| + |\mathcal{V}|))^{k_1+k_2})$ time and $O(n + |\mathcal{U}| + |\mathcal{V}|)$ space, with pairs $(\mathcal{U}', \mathcal{V}')$ generated one by one and each set $\mathcal{U}' \cup \mathcal{V}'$ sorted in two ways: according to the lexicographic order of the modified strings on the first and the second coordinate, respectively.*

Proof. We use the construction from Proposition 7.15. We apply the construction of a bicomplete family (Lemma 7.12) to generate the family $\mathbf{G}$ in batches $\mathbf{G}'$ consisting of sets of total size $\Theta(n + |\mathcal{G}|)$ (the last batch might be smaller). In the algorithm $\mathbf{G}$ is processed in batches of size $\Omega(n)$ (and not, say, set by set) so that the time required to bucket sort integers of magnitude $n^{O(1)}$ in the algorithm does not make the overall time complexity worse. For each batch $\mathbf{G}' \subseteq \mathbf{G}$, we first construct all the non-empty sets $\mathcal{G}'_{\delta_1,\delta_2}$ for $\mathcal{G}' \in \mathbf{G}'$. For this, we iterate over sets $\mathcal{G}' \in \mathbf{G}'$, pairs $(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{G}'$, subsets $\delta_1 \subseteq \Delta_1$ and subsets $\delta_2 \subseteq \Delta_2$, creating 5-tuples consisting of the index of $\mathcal{G}'$ in $\mathbf{G}'$, as well as δ_1, δ_2, $F_1^{\Delta_1}$ and $F_2^{\Delta_2}$. We group these 5-tuples according to the first three coordinates; the key consists of $O(1 + k_1 + k_2) = O(1)$ integers bounded by $n^{O(1)}$, so we use radix sort. Moreover, since radix sort is stable, the sets $\mathcal{G}'_{\delta_1,\delta_2}$ can be constructed along with both linear orders derived from $\mathcal{G}'$. Finally, for each non-empty set $\mathcal{G}'_{\delta_1,\delta_2}$ with $\mathcal{G}' \in \mathbf{G}'$, we iterate over $d_1 \in [0\,..\,k_1]$ and $d_2 \in [0\,..\,k_2]$, generating the subsets $\mathcal{U}'$ and $\mathcal{V}'$ of $\mathcal{G}'_{\delta_1,\delta_2}$ according to the formulae above. The two linear orders of $\mathcal{U}' \cup \mathcal{V}'$ are derived from $\mathcal{G}'_{\delta_1,\delta_2}$.

We conclude with the complexity analysis. The construction of the (k_1,k_2)-bicomplete family of Lemma 7.12 requires $O(n + |\mathcal{G}|\min(\ell, \log|\mathcal{G}|)^{k_1+k_2})$ time and $O(n + |\mathcal{G}|)$ space. The total size of the sets in each batch $\mathbf{G}'$ is $O(n + |\mathcal{G}|)$ and the number of batches is $O(1 + \frac{|\mathcal{G}|}{n+|\mathcal{G}|}\min(\ell, \log|\mathcal{G}|)^{k_1+k_2})$. For

each modified string $(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{G}'$, we construct $2^{|\Delta_1|+|\Delta_2|} \le 2^{k_1+k_2} = O(1)$ tuples corresponding to elements of the sets $\mathcal{G}'_{\delta_1,\delta_2}$, in $O(1)$ time per tuple. Consequently, this phase uses $O(n + |\mathcal{G}|)$ space and $O(|\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1+k_2})$ time. For each batch, we use $O(n + |\mathcal{G}|)$ time and space for radix sort, yielding a total of $O(n + |\mathcal{G}|)$ space and $O(n + |\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1+k_2})$ time. Finally, for each non-empty set $\mathcal{G}'_{\delta_1,\delta_2}$ with $\mathcal{G}' \in \mathbf{G}$, each $d_1 \in [0\,..\,k_1]$ and each $d_2 \in [0\,..\,k_2]$, we spend $O(|\mathcal{G}'_{\delta_1,\delta_2}|)$ time and space to generate $\mathcal{U}'$ and $\mathcal{V}'$. Since $k_1, k_2 = O(1)$, the overall time of this final phase is proportional to the total size of the sets $\mathcal{G}'_{\delta_1,\delta_2}$, which is $O(|\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1+k_2})$ as argued in the proof of Proposition 7.15. □

We are ready to provide a proof of Theorem 6.6. With condition (4) of Proposition 7.15, instead of computing $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V})$, it suffices to compute $\max_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}} \mathrm{maxPairLCP}(\mathcal{U}', \mathcal{V}')$. We process the family $\mathbf{P}$ in batches of size $\Omega(N)$ to use the fact that the complexity in Lemma 1.5 has a denominator $\log N$ (instead of, say, a logarithm of the size of a set in $\mathbf{P}$). Still, the batch size is bounded by $O(N)$ to ensure small space complexity. By Lemma 7.18, there are $O(\min(\ell, \log N)^k)$ batches. For each of them we create an instance of Two String Families LCP Problem, which we solve using Lemma 1.3 if $\ell > \log^{3/2} N$ or Lemma 1.5 otherwise to obtain the desired complexity.

Let us restate the theorem for convenience.

Theorem 6.6. *Consider two (ℓ, ℓ)-families $\mathcal{U}, \mathcal{V}$ of total size N consisting of pairs of substrings of a given length-n text over the alphabet $[0\,..\,n)$. For any non-negative integers $k_1, k_2 = O(1)$, the value $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V})$ can be computed:*

- *in $O(n + N \log^{k_1+k_2+1} N)$ time and $O(n + N)$ space if $\ell > \log^{3/2} N$,*
- *in $O(n + N\ell \log^{k_1+k_2-1/2} N)$ time and $O(n + N\ell/\log N)$ space if $\log N < \ell \le \log^{3/2} N$,*
- *in $O(n + N\ell^{k_1+k_2}\sqrt{\log N})$ time and $O(n + N)$ space if $\ell \le \log N$.*

Proof. First, we augment the given text in $O(n)$ time with a data structure for $O(1)$-time LCP queries (see Theorem 2.10). Next, we apply Proposition 7.15 and 7.18 to generate a family $\mathbf{P}$ such that $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V}) = \max_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}} \mathrm{maxPairLCP}(\mathcal{U}', \mathcal{V}')$. We process pairs $(\mathcal{U}', \mathcal{V}') \in \mathbf{P}$ in batches $\mathbf{P}' \subseteq \mathbf{P}$ consisting of pairs of sets of total size $\Theta(N)$ (in the last batch, the total size might be smaller). Each batch $\mathbf{P}' = \{(\mathcal{U}'_j, \mathcal{V}'_j) : j \in [1\,..\,p]\}$ is processed as follows.

First, for each $j \in [1\,..\,p]$, we construct the compacted trie $\mathcal{T}(\mathcal{F}_j)$ of the set $\mathcal{F}_j = \mathcal{F}_{1,j} \cup \mathcal{F}_{2,j}$, where:

$$\mathcal{F}_{i,j} = \{F_i^{\Delta_i} : (F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{U}'_j \cup \mathcal{V}'_j\}.$$

The two linear orders associated with the set $\mathcal{U}'_j \cup \mathcal{V}'_j$ yield the lexicographic order of the sets $\mathcal{F}_{1,j}$ and $\mathcal{F}_{2,j}$. These orders can be merged to a linear order of $\mathcal{F}_j$ using a classic linear-time algorithm with the aid of LCP queries used to compare any two modified substrings. In order to build the compacted tries $\mathcal{T}(\mathcal{F}_j)$ using Lemma 2.3, it suffices to determine the longest common prefixes between any two consecutive modified strings in $\mathcal{F}_j$, which reduces to LCP queries on the text.

Then we construct the compacted trie $\mathcal{T}(\mathcal{F})$ of the set:

$$\mathcal{F} = \bigcup_{j=1}^{p} \{j \cdot F^{\Delta} : F^{\Delta} \in \mathcal{F}_j\} \subseteq [1\,..\,p] \cdot \Sigma^{\le \ell}.$$

For this, we create a new root node and, for $j \in [1\,..\,m]$, attach the compacted trie $\mathcal{T}(\mathcal{F}_j)$ with a length-1 edge; if the root of $\mathcal{T}(\mathcal{F}_j)$ has degree 1 and $\varepsilon \notin \mathcal{F}_j$, we also dissolve the root. Finally, we

construct two sets $\mathcal{P}, \mathcal{Q} \subseteq \mathcal{F}^2$:

$$\mathcal{P} = \bigcup_{j=1}^{p} \{(j \cdot U_1^{\Delta_1}, j \cdot U_2^{\Delta_2}) : (U_1^{\Delta_1}, U_2^{\Delta_2}) \in \mathcal{U}'_j\},$$

$$\mathcal{Q} = \bigcup_{j=1}^{p} \{(j \cdot V_1^{\nabla_1}, j \cdot V_2^{\nabla_2}) : (V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{V}'_j\}.$$

This yields an instance of the Two String Families LCP Problem, which we solve using Lemma 1.3 if $\ell > \log^{3/2} N$ or Lemma 1.5 otherwise. The obtained value satisfies $\mathrm{maxPairLCP}(\mathcal{P}, \mathcal{Q}) = 2 + \max_{j=1}^{m} \mathrm{maxPairLCP}(\mathcal{U}'_j, \mathcal{V}'_j)$ and, taking the maximum across all batches $\mathbf{P}' \subseteq \mathbf{P}$, we retrieve $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V}) = \max_{(\mathcal{U}',\mathcal{V}') \in \mathbf{P}} \mathrm{maxPairLCP}(\mathcal{U}', \mathcal{V}')$ (cf. condition (4) of Proposition 7.15).

We conclude with the complexity analysis. Applying the algorithm of Lemma 7.18 costs $O(n + N \min(\ell, \log N)^{k_1+k_2})$ time and $O(n+N)$ space. According to Proposition 7.15, the resulting family $\mathbf{P}$ satisfies $\max_{(\mathcal{U}',\mathcal{V}') \in \mathbf{P}}(|\mathcal{U}'|+|\mathcal{V}'|) = O(N)$ and $\sum_{(\mathcal{U}',\mathcal{V}') \in \mathbf{P}} = O(N \min(\ell, \log N)^{k_1+k_2})$. Hence, the number of batches $\mathbf{P}'$ is $O(\min(\ell, \log N)^{k_1+k_2})$. Let us now focus on a single batch $\mathbf{P}' = \{(\mathcal{U}'_j, \mathcal{V}'_j) : 1 \le j \le p\}$; recall that it satisfies $\sum_{j=1}^{p}(|\mathcal{U}'_j| + |\mathcal{V}'_j|) = O(N)$. Each set $\mathcal{F}_j$ is obtained from $\mathcal{U}'_j \cup \mathcal{V}'_j$ in $O(|\mathcal{U}'_j| + |\mathcal{V}'_j|)$ time. Each modified string in $\mathcal{F}_j$ contains up to $k_1 + k_2$ modifications, so the LCP computation for two such modified strings requires up to $2(k_1 + k_2) + 1 = O(1)$ LCP queries on T. Consequently, each trie $\mathcal{T}(\mathcal{F}_j)$ is constructed in $O(|\mathcal{U}'_j| + |\mathcal{V}'_j|)$ time. Merging these tries into $\mathcal{T}(\mathcal{F})$ requires $O(p)$ additional time, i.e., the construction of $\mathcal{T}(\mathcal{F})$ takes $O(N)$ time in total. The sets $\mathcal{P}$ and $\mathcal{Q}$ are of size $O(N)$ and also take $O(N)$ time to construct. Overall, preparing the instance $(\mathcal{T}(\mathcal{F}), \mathcal{P}, \mathcal{Q})$ of Two String Families LCP Problem takes $O(N)$ time and space. Solving this instance takes $O(N \log N)$ time and $O(N)$ space if we use Lemma 1.3 or $O(N(\ell + \log N)(\log \ell + \sqrt{\log N})/\log N)$ time and $O(N + N\ell/\log N)$ space if we use Lemma 1.5. In either case, this final step dominates both the time and the space complexity of processing a single batch $\mathbf{P}'$. Across all $O(\min(\ell, \log N)^{k_1+k_2})$ batches, we obtain the following trade-offs:

- $O(N \log N \cdot \log^{k_1+k_2} N)$ time and $O(N)$ space if $\ell > \log^{3/2} N$;
- $O((N\ell/\sqrt{\log N}) \cdot \log^{k_1+k_2} N)$ time and $O(N\ell/\log N)$ space if $\log N < \ell \le \log^{3/2} N$;
- $O(N\sqrt{\log N} \cdot \ell^{k_1+k_2})$ time and $O(N)$ space if $\ell \le \log N$.

Accounting for $O(n)$ space and construction time of the data structure for LCP queries on the text, we retrieve the claimed trade-offs. □

References

[1] Amir Abboud, Richard Ryan Williams, and Huacheng Yu. 2015. More applications of the polynomial method to algorithm design. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '15)*. SIAM, 218–230. DOI: https://doi.org/10.1137/1.9781611973730.17

[2] Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan. 2022. The heaviest induced ancestors problem: Better data structures and applications. *Algorithmica* 84, 7 (2022), 2088–2105. DOI: https://doi.org/10.1007/S00453-022-00955-7

[3] Shyan Akmal and Ce Jin. 2023. Near-optimal quantum algorithms for string problems. *Algorithmica* 85, 8 (2023), 2260–2317. DOI: https://doi.org/10.1007/S00453-022-01092-X

[4] Amihood Amir and Itai Boneh. 2018. Locally maximal common factors as a tool for efficient dynamic string algorithms. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM '18)*. LIPIcs, Vol. 105, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 11, 1–13. DOI: https://doi.org/10.4230/LIPIcs.CPM.2018.11

[5] Amihood Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. 2017. Longest common factor after one edit operation. In *Proceedings of the 24th International Symposium on String*

Processing and Information Retrieval (SPIRE '17). Lecture Notes in Computer Science, Vol. 10508, Springer, 14–26. DOI: https://doi.org/10.1007/978-3-319-67428-5_2

[6] Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. 2020. Dynamic and internal longest common substring. *Algorithmica* 82, 12 (2020), 3707–3743. DOI: https://doi.org/10.1007/s00453-020-00744-0

[7] Arne Andersson and Mikkel Thorup. 2007. Dynamic ordered sets with exponential search trees. *Journal of the ACM* 54, 3 (2007), 13. DOI: https://doi.org/10.1145/1236457.1236460

[8] Lorraine A. K. Ayad, Carl Barton, Panagiotis Charalampopoulos, Costas S. Iliopoulos, and Solon P. Pissis. 2018. Longest common prefixes with k-errors and applications. In *Proceedings of the 25th International Symposium on String Processing and Information Retrieval (SPIRE '18)*. Lecture Notes in Computer Science, Vol. 11147, Springer, 27–41. DOI: https://doi.org/10.1007/978-3-030-00479-8_3

[9] Maxim A. Babenko, Paweł Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. 2015. Wavelet trees meet suffix trees. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '15)*, 572–591. DOI: https://doi.org/10.1137/1.9781611973730.39

[10] Maxim A. Babenko and Tatiana Starikovskaya. 2011. Computing the longest common substring with one mismatch. *Problems of Information Transmission* 47, 1 (2011), 28–33. DOI: https://doi.org/10.1134/S0032946011010030

[11] Ricardo A. Baeza-Yates. 1989. Improved string searching. *Software: Practice and Experience* 19, 3 (1989), 257–271. DOI: https://doi.org/10.1002/spe.4380190305

[12] Hideo Bannai and Jonas Ellert. 2023. Lyndon arrays in sublinear time. In *Proceedings of the 31st Annual European Symposium on Algorithms (ESA '23)*. LIPIcs, Vol. 274, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 14, 1–16. DOI: https://doi.org/10.4230/LIPIcs.ESA.2023.14

[13] Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. 2017. The "runs" theorem. *SIAM Journal on Computing* 46, 5 (2017), 1501–1514. DOI: https://doi.org/10.1137/15M1011032

[14] Jérémy Barbay, Francisco Claude, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. 2014. Efficient fully-compressed sequence representations. *Algorithmica* 69, 1 (2014), 232–268. DOI: https://doi.org/10.1007/S00453-012-9726-3

[15] Djamal Belazzougui. 2010. Worst case efficient single and multiple string matching in the RAM model. In *Proceedings of the 21st International Workshop on Combinatorial Algorithms (IWOCA '10)*. Lecture Notes in Computer Science, Vol. 6460, Springer, 90–102. DOI: https://doi.org/10.1007/978-3-642-19222-7_10

[16] Djamal Belazzougui. 2015. Improved space-time tradeoffs for approximate full-text indexing with one edit error. *Algorithmica* 72, 3 (2015), 791–817. DOI: https://doi.org/10.1007/s00453-014-9873-9

[17] Djamal Belazzougui and Gonzalo Navarro. 2014. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms* 10, 4, Article 23 (2014), 1–19. DOI: https://doi.org/10.1145/2635816

[18] Djamal Belazzougui and Gonzalo Navarro. 2015. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms* 11, 4, Article 31 (2015), 1–21. DOI: https://doi.org/10.1145/2629339

[19] Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gą sieniec, Roberto Grossi, and Oren Weimann. 2014. Towards optimal packed string matching. *Theoretical Computer Science* 525 (2014), 111–129. DOI: https://doi.org/10.1016/j.tcs.2013.06.013

[20] Stav Ben-Nun, Shay Golan, Tomasz Kociumaka, and Matan Kraus. 2020. Time-space tradeoffs for finding a long common substring. In *Proceedings of the 31st Annual Symposium on Combinatorial Pattern Matching (CPM '20)*. LIPIcs, Vol. 161, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 5, 1–14. DOI: https://doi.org/10.4230/LIPIcs.CPM.2020.5

[21] Michael A. Bender and Martin Farach-Colton. 2000. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN '00)*. Lecture Notes in Computer Science, Vol. 1776, Springer, 88–94. DOI: https://doi.org/10.1007/10719839_9

[22] Philip Bille. 2009. Fast searching in packed strings. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM '09)*. Lecture Notes in Computer Science, Vol. 5577, Springer, 116–126. DOI: https://doi.org/10.1007/978-3-642-02441-2_11

[23] Philip Bille. 2011. Fast searching in packed strings. *Journal of Discrete Algorithms* 9, 1 (2011), 49–56. DOI: https://doi.org/10.1016/j.jda.2010.09.003

[24] Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. 2017. Deterministic indexing for packed strings. In *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM '17)*. LIPIcs, Vol. 78, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 6, 1–11. DOI: https://doi.org/10.4230/LIPIcs.CPM.2017.6

[25] Dany Breslauer, Leszek Gasieniec, and Roberto Grossi. 2012. Constant-time word-size string matching. In *Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM '12)*. Lecture Notes in Computer Science, Vol. 7354, 83–96. DOI: https://doi.org/10.1007/978-3-642-31265-6_7

[26] Karl Bringmann, Philip Wellnitz, and Marvin Künnemann. 2019. Few matches or almost periodicity: Faster pattern matching with mismatches in compressed texts. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '19)*. SIAM, 1126–1145. DOI: https://doi.org/10.1137/1.9781611975482.69

[27] Gerth Stølting Brodal and Christian N. S. Pedersen. 2000. Finding maximal quasiperiodicities in strings. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM '00)*. Lecture Notes in Computer Science, Vol. 1848, 397–411. DOI: https://doi.org/10.1007/3-540-45123-4_33

[28] Mark R. Brown and Robert E. Tarjan. 1979. A fast merging algorithm. *Journal of the ACM* 26, 2 (1979), 211–226. DOI: https://doi.org/10.1145/322123.322127

[29] Stefan Burkhardt and Juha Kärkkäinen. 2003. Fast lightweight suffix array construction and checking. In *Proceedings of the Combinatorial Pattern Matching (CPM '03)*. LNCS, Vol. 2676, Springer, 55–69. DOI: https://doi.org/10.1007/3-540-44888-8_5

[30] Domenico Cantone and Simone Faro. 2009. Pattern matching with swaps for short patterns in linear time. In *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM '09)*. Lecture Notes in Computer Science, Vol. 5404, Springer, 255–266. DOI: https://doi.org/10.1007/978-3-540-95891-8_25

[31] Ho-Leung Chan, Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. 2010. Compressed indexes for approximate string matching. *Algorithmica* 58, 2 (2010), 263–281. DOI: https://doi.org/10.1007/s00453-008-9263-2

[32] Ho-Leung Chan, Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. 2011. A linear size index for approximate pattern matching. *Journal of Discrete Algorithms* 9, 4 (2011), 358–364. DOI: https://doi.org/10.1016/j.jda.2011.04.004

[33] Timothy M. Chan and Mihai Pătraşcu. 2010. Counting inversions, offline orthogonal range counting, and related problems. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '10)*. SIAM, 161–173. DOI: https://doi.org/10.1137/1.9781611973075.15

[34] Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. 2018. Linear-time algorithm for long LCF with k mismatches. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM '18)*, Article 23, 1–16. DOI: https://doi.org/10.4230/LIPIcs.CPM.2018.23

[35] Panagiotis Charalampopoulos, Bartlomiej Dudek, Pawel Gawrychowski, and Karol Pokorski. 2023. Optimal near-linear space heaviest induced ancestors. In *Proceedings of the 34th Annual Symposium on Combinatorial Pattern Matching (CPM '23)*. LIPIcs, Vol. 259, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 8, 1–18. DOI: https://doi.org/10.4230/LIPICS.CPM.2023.8

[36] Panagiotis Charalampopoulos, Paweł Gawrychowski, and Karol Pokorski. 2020. Dynamic longest common substring in polylogarithmic time. In *Proceedings of the 47th International Colloquium on Automata, Languages, and Programming (ICALP '20)*. LIPIcs, Vol. 168, Article 27, 1–19. DOI: https://doi.org/10.4230/LIPIcs.ICALP.2020.27

[37] Panagiotis Charalampopoulos, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, and Juliusz Straszyński. 2022. Efficient computation of sequence mappability. *Algorithmica* 84, 5 (2022), 1418–1440. DOI: https://doi.org/10.1007/S00453-022-00934-Y

[38] Panagiotis Charalampopoulos, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis. 2020. Property suffix array with applications in indexing weighted sequences. *Journal of Experimental Algorithmics* 25 (2020), 1–16. DOI: https://doi.org/10.1145/3385898

[39] Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. 2021. Faster algorithms for longest common substring. In *Proceedings of the 29th Annual European Symposium on Algorithms (ESA '21)*. LIPIcs, Vol. 204, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 30, 1–17. DOI: https://doi.org/10.4230/LIPIcs.ESA.2021.30

[40] Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. 2021. Circular pattern matching with k mismatches. *Journal of Computer and System Sciences* 115 (2021), 73–85. DOI: https://doi.org/10.1016/j.jcss.2020.07.003

[41] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. 2020. Faster approximate pattern matching: A unified approach. In *Proceedings of the 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS '20)*. IEEE, 978–989. DOI: https://doi.org/10.1109/FOCS46700.2020.00095

[42] Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. 2022. Longest palindromic substring in sublinear time. In *Proceedings of the 33rd Annual Symposium on Combinatorial Pattern Matching (CPM '22)*. LIPIcs, Vol. 223, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 20, 1–9. DOI: https://doi.org/10.4230/LIPIcs.CPM.2022.20

[43] Archie L. Cobbs. 1995. Fast approximate matching using suffix trees. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM '95). Lecture Notes in Computer Science, Vol.* 937, 41–54. DOI: https://doi.org/10.1007/3-540-60044-2_33

[44] Vincent Cohen-Addad, Laurent Feuilloley, and Tatiana Starikovskaya. 2019. Lower bounds for text indexing with mismatches and differences. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '19)*, 1146–1164. DOI: https://doi.org/10.1137/1.9781611975482.70

[45] Charles J. Colbourn and Alan C. H. Ling. 2000. Quorums from difference covers. *Information Processing Letters* 75, 1–2 (2000), 9–12. DOI: https://doi.org/10.1016/S0020-0190(00)00080-6

[46] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. 2004. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC '04)*, 91–100. DOI: https://doi.org/10.1145/1007352.1007374

[47] Richard Cole and Ramesh Hariharan. 2002. Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing* 31, 6 (2002), 1761–1782. DOI: https://doi.org/10.1137/S0097539700370527

[48] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. 2007. *Algorithms on Strings*. Cambridge University Press. DOI: https://doi.org/10.1017/cbo9780511546853

[49] Maxime Crochemore, Costas S. Iliopoulos, Manal Mohamed, and Marie-France Sagot. 2006. Longest repeats with a block of k don't cares. *Theoretical Computer Science* 362, 1–3 (2006), 248–254. DOI: https://doi.org/10.1016/j.tcs.2006.06.029

[50] Jonas Ellert. 2023. Sublinear time Lempel-Ziv (LZ77) factorization. In *Proceedings of the 30th International Symposium on String Processing and Information Retrieval (SPIRE '23)*. Lecture Notes in Computer Science, Vol. 14240, 171–187. DOI: https://doi.org/10.1007/978-3-031-43980-3_14

[51] Jonas Ellert and Johannes Fischer. 2021. Linear time runs over general ordered alphabets. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP '21)*. LIPIcs, Vol. 198, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 63, 1–16. DOI: https://doi.org/10.4230/LIPIcs.ICALP.2021.63

[52] Martin Farach. 1997. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, 137–143. DOI: https://doi.org/10.1109/SFCS.1997.646102

[53] Nathan J. Fine and Herbert S. Wilf. 1965. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society* 16, 1 (1965), 109–114. DOI: https://doi.org/10.1090/S0002-9939-1965-0174934-9

[54] Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. 2015. Longest common substrings with k mismatches. *Information Processing Letters* 115, 6–8 (2015), 643–647. DOI: https://doi.org/10.1016/j.ipl.2015.03.006

[55] Kimmo Fredriksson. 2002. Faster string matching with super-alphabets. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE '02)*. Lecture Notes in Computer Science, Vol. 2476, Springer, 44–57. DOI: https://doi.org/10.1007/3-540-45735-6_5

[56] Kimmo Fredriksson. 2003. Shift-or string matching with super-alphabets. *Information Processing Letters* 87, 4 (2003), 201–204. DOI: https://doi.org/10.1016/S0020-0190(03)00296-5

[57] Harold N. Gabow and Robert Endre Tarjan. 1985. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences* 30, 2 (1985), 209–221. DOI: https://doi.org/10.1016/0022-0000(85)90014-5

[58] Travis Gagie, Pawel Gawrychowski, and Yakov Nekrich. 2013. Heaviest induced ancestors and longest common substrings. In *Proceedings of the 25th Canadian Conference on Computational Geometry (CCCG '13)*. Carleton University, Ottawa, Canada. Retrieved from http://cccg.ca/proceedings/2013/papers/paper_29.pdf

[59] Emanuele Giaquinta, Szymon Grabowski, and Kimmo Fredriksson. 2013. Approximate pattern matching with k-mismatches in packed text. *Information Processing Letters* 113, 19–21 (2013), 693–697. DOI: https://doi.org/10.1016/j.ipl.2013.07.002

[60] Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya. 2020. Approximating longest common substring with k mismatches: Theory and practice. In *Proceedings of the 31st Annual Symposium on Combinatorial Pattern Matching (CPM '20)*. LIPIcs, Vol. 161, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 16, 1–15. DOI: https://doi.org/10.4230/LIPICS.CPM.2020.16

[61] Szymon Grabowski. 2015. A note on the longest common substring with k-mismatches problem. *Information Processing Letters* 115, 6–8 (2015), 640–642. DOI: https://doi.org/10.1016/j.ipl.2015.03.003

[62] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2003. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 841–850. Retrieved from http://dl.acm.org/citation.cfm?id=644108.644250

[63] Yijie Han. 2004. Deterministic sorting in $O(n \log\log n)$ time and linear space. *Journal of Algorithms* 50, 1 (2004), 96–105. DOI: https://doi.org/10.1016/j.jalgor.2003.09.001

[64] Lucas Chi and Kwong Hui. 1992. Color set size problem with application to string matching. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM '92)*. Lecture Notes in Computer Science, Vol. 644, 230–243. DOI: https://doi.org/10.1007/3-540-56024-6_19

[65] Trinh N. D. Huynh, Wing-Kai Hon, Tak Wah Lam, and Wing-Kin Sung. 2006. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science* 352, 1–3 (2006), 240–249. DOI: https://doi.org/10.1016/j.tcs.2005.11.022

[66] Russell Impagliazzo and Ramamohan Paturi. 2001. On the complexity of k-SAT. *Journal of Computer and System Sciences* 62, 2 (2001), 367–375. DOI: https://doi.org/10.1006/jcss.2000.1727

[67] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. 2001. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences* 63, 4 (2001), 512–530. DOI: https://doi.org/10.1006/jcss.2001.1774

[68] Ce Jin and Jakob Nogler. 2024. Quantum speed-ups for string synchronizing sets, longest common substring, and k-mismatch matching. *ACM Transactions on Algorithms* 20, 4, Article 32 (2024), 1–36. DOI: https://doi.org/10.1145/3672395

[69] Dominik Kempa and Tomasz Kociumaka. 2019. String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC '19)*. ACM, 756–767. DOI: https://doi.org/10.1145/3313276.3316368

[70] Dominik Kempa and Tomasz Kociumaka. 2025. On the hardness hierarchy for the $O(n\sqrt{\log n})$ complexity in the word RAM. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing (STOC '25)*. ACM, 290–300. DOI: https://doi.org/10.1145/3717823.3718291

[71] Shmuel Tomi Klein and Miri Ben-Nissan. 2007. Accelerating Boyer Moore searches on binary texts. In *Implementation and Application of Automata, 12th International Conference (CIAA '07)*. Lecture Notes in Computer Science, Vol. 4783, Springer, 130–143. DOI: https://doi.org/10.1007/978-3-540-76336-9_14

[72] Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya. 2019. Longest common substring with approximately k mismatches. *Algorithmica* 81, 6 (2019), 2633–2652. DOI: https://doi.org/10.1007/s00453-019-00548-x

[73] Tomasz Kociumaka, Tatiana Starikovskaya, and Hjalte Wedel Vildhøj. 2014. Sublinear space algorithms for the longest common substring problem. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA '14)*. Lecture Notes in Computer Science, Vol. 8737, 605–617. DOI: https://doi.org/10.1007/978-3-662-44777-2_50

[74] Roman M. Kolpakov and Gregory Kucherov. 1999. Finding maximal repetitions in a word in linear time. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS '99)*, 596–604. DOI: https://doi.org/10.1109/SFFCS.1999.814634

[75] George Lagogiannis, Christos Makris, and Athanasios K. Tsakalidis. 2006. Reducing structural changes in van Emde Boas' data structure to the lower bound for the dynamic predecessor problem. *Journal of Discrete Algorithms* 4, 1 (2006), 106–141. DOI: https://doi.org/10.1016/J.JDA.2005.01.006

[76] François Le Gall and Saeed Seddighin. 2023. Quantum meets fine-grained complexity: Sublinear time quantum algorithms for string problems. *Algorithmica* 85, 5 (2023), 1251–1286. DOI: https://doi.org/10.1007/S00453-022-01066-Z

[77] Mamoru Maekawa. 1985. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems* 3, 2 (1985), 145–159. DOI: https://doi.org/10.1145/214438.214445

[78] Udi Manber and Eugene W. Myers. 1993. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing* 22, 5 (1993), 935–948. DOI: https://doi.org/10.1137/0222058

[79] Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, and Kazuo Hashimoto. 2009. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theoretical Computer Science* 410, 8–10 (2009), 900–913. DOI: https://doi.org/10.1016/J.TCS.2008.12.016

[80] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. 2017. Space-efficient construction of compressed indexes in deterministic linear time. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '17)*. SIAM, 408–424. DOI: https://doi.org/10.1137/1.9781611974782.26

[81] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. 2020. Fast compressed self-indexes with deterministic linear-time construction. *Algorithmica* 82, 2 (2020), 316–337. DOI: https://doi.org/10.1007/S00453-019-00637-X

[82] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. 2020. Text indexing and searching in sublinear time. In *Proceedings of the 31st Annual Symposium on Combinatorial Pattern Matching (CPM '20)*. LIPIcs, Vol. 161, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 24, 1–15. DOI: https://doi.org/10.4230/LIPIcs.CPM.2020.24

[83] J. Ian Munro, Yakov Nekrich, and Jeffrey S. Vitter. 2016. Fast construction of wavelet trees. *Theoretical Computer Science* 638 (2016), 91–97. DOI: https://doi.org/10.1016/j.tcs.2015.11.011

[84] Gonzalo Navarro and Yakov Nekrich. 2017. Time-optimal top-k document retrieval. *SIAM Journal on Computing* 46, 1 (2017), 80–113. DOI: https://doi.org/10.1137/140998949

[85] Gonzalo Navarro and Mathieu Raffinot. 1998. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching (CPM '98)*. Lecture Notes in Computer Science, Vol. 1448, Springer, 14–33. DOI: https://doi.org/10.1007/BFb0030778

[86] Tatiana Starikovskaya and Hjalte Wedel Vildhøj. 2013. Time-space trade-offs for the longest common substring problem. In *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM '13)*, 223–234. DOI: https://doi.org/10.1007/978-3-642-38905-4_22

[87] Jorma Tarhio and Hannu Peltola. 1997. String matching in the DNA alphabet. *Software: Practice and Experience* 27, 7 (1997), 851–861. Retrieved from https://doi.org/10.1002/(SICI)1097-024X(199707)27:7$<$851::AID-SPE108$>$3.0.CO;2-D

[88] Sharma V. Thankachan, Chaitanya Aluru, Sriram P. Chockalingam, and Srinivas Aluru. 2018. Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In *Proceedings of the 22nd Annual International Conference on Research in Computational Molecular Biology (RECOMB '18)*. Lecture Notes in Computer Science, Vol. 10812, Springer, 211–224. DOI: https://doi.org/10.1007/978-3-319-89929-9_14

[89] Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. 2016. A provably efficient algorithm for the k-mismatch average common substring problem. *Journal of Computational Biology* 23, 6 (2016), 472–482. DOI: https://doi.org/10.1089/cmb.2015.0235

[90] Dekel Tsur. 2010. Fast index for approximate string matching. *Journal of Discrete Algorithms* 8, 4 (2010), 339–345. DOI: https://doi.org/10.1016/j.jda.2010.08.002
[91] Esko Ukkonen. 1993. Approximate string-matching over suffix trees. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM '93)*, 228–242. DOI: https://doi.org/10.1007/BFb0029808
[92] Peter van Emde Boas. 1977. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters* 6, 3 (1977), 80–82. DOI: https://doi.org/10.1016/0020-0190(77)90031-X
[93] Peter Weiner. 1973. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, 1–11. DOI: https://doi.org/10.1109/SWAT.1973.13
[94] Dan E. Willard. 1983. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters* 17, 2 (1983), 81–84. DOI: https://doi.org/10.1016/0020-0190(83)90075-3
[95] Dan E. Willard. 2000. Examining computational geometry, Van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing* 29, 3 (2000), 1030–1049. DOI: https://doi.org/10.1137/S0097539797322425

Appendices

A Proof of Proposition 2.4

Throughout the appendix, we restate the propositions/lemmas for convenience.

Proposition 2.4. *We consider a deterministic streaming algorithm* A *that reads* $O(1)$ *input streams, writes* $O(1)$ *output streams, uses* s *bits of working space and executes at most* t *instructions (each in* $O(1)$ *time) between subsequent read/write instructions. The streams are represented in a packed form with word size* w*. For every integer* $\tau \in [1\,..\,w]$*, if* $s \leq w$*, after an* $O(2^{\tau+s}(\tau t + s))$*-time preprocessing, any execution of* A *can be simulated in* $O(1 + L/\tau)$ *time, where* L *is the total length (in bits) of all input and output streams.*

Proof. The sequence of computations in the algorithm depends only on the next instruction to be performed, its current memory state and the data that appear on the input streams. More precisely, in the preprocessing:

- for every possible instruction in the algorithm's code ($O(1)$ options; we can assume that it is a single-bit read or write instruction or the first instruction of the algorithm),
- for every possible state of memory used in the algorithm (2^s options) and
- for every possible combination of the next (up to) $\lfloor \tau/x \rfloor$ bits that are available on each of the input streams, where x is the number of input streams ($O(2^\tau)$ possible combinations),

the algorithm is simulated starting from the selected instruction with the given memory state. The simulation lasts until at least one of the following events takes place:

- the algorithm terminates or
- the algorithm requires to read a bit of an input stream that is not given to it (that is, the $(\lfloor \tau/x \rfloor + 1)$th bit of a stream) or
- the algorithm requires to write the $(\tau + 1)$th bit.

The computations take $O(\tau t)$ time. When the simulation ends, the following data is memoised:

- the (read or write) instruction on which the simulation ended, or information that the algorithm terminated ($O(1)$ options),
- the final state of the algorithm's memory (s bits),
- the number of bits read from each of the input streams ($O(x \log \tau) = O(\log \tau)$ bits) and
- the number of bits written to each of the output streams and the bits written to all output streams, ordered by the stream and then in the order of writing ($O(y \log \tau + \tau) = O(\tau)$ bits, where y is the number of output streams).

In total, in the preprocessing, an array consisting of $O(2^{s+\tau})$ cells is constructed. Each cell is represented using $O(1 + (\tau + s)/w) = O(1)$ machine words. The total preprocessing time is $O(2^{\tau+s}(\tau t + s))$, as required.

The constructed array allows us to simulate the algorithm in $O(1 + Lx/\tau) = O(1 + L/\tau)$ time, where L is the total length (in bits) of all input and output streams, in a straightforward way. In the simulation, we store:

- —the current instruction to be executed in the algorithm,
- —the current state of the algorithm's memory,
- —for each input stream, the position of the next bit to be read and
- —the data on the output streams.

In each step of the simulation, $O(\tau)$ bits of the input are processed or $O(\tau)$ bits are output. Each step is performed in $O(1)$ time thanks to the fact that the streams are given in a packed form and $s, \tau \leq w$. Hence the total processing complexity. □

B Proof of Lemma 2.8

Lemma 2.8. *For a positive integer τ, a string $T \in [0\,..\,\sigma)^n$ contains $O(n/\tau)$ τ-runs. Moreover, if $\tau \leq \frac{1}{4}\log_\sigma n$, given a packed representation of T, we can compute all τ-runs in T and group them by their Lyndon roots in $O(n/\tau)$ time. Within the same complexities, for each τ-run, we can compute the two leftmost occurrences of its Lyndon root.*

Proof. The first claim of the lemma follows from the periodicity lemma (Lemma 2.2). Recall that a τ-run is a run of length at least $3\tau - 1$ with period at most $\frac{1}{3}\tau$. Suppose, towards a contradiction, that two different τ-runs overlap by more than $\frac{2}{3}\tau$ positions. By the fact that their periods are at most $\frac{1}{3}\tau$ and an application of Lemma 2.2, we obtain that these two τ-runs cannot be distinct as they share the same smallest period; a contradiction. This implies that two distinct τ-runs can overlap by no more than $\frac{2}{3}\tau$ positions. In turn, this implies that T contains $O(n/\tau)$ runs.

Let us now show how to efficiently compute τ-runs. We first compute a τ-synchronising set A in $O(n/\tau)$ time using Theorem 2.5. By the definition of such a set, a position $i \in [1\,..\,n - 3\tau + 2]$ in T is a starting position of a τ-run if and only if $[i\,..\,i + \tau) \cap A = \emptyset$ and $i - 1 \in A$. The period of this τ-run is equal to $p = \mathrm{per}(T[i\,..\,i + 3\tau - 2])$. Then, by Lemma 2.6, the longest prefix of $T[i\,..\,n]$ with period p is $T[i\,..\,\mathrm{succ}_A(i) + 2\tau - 2]$. Using these conditions, we can compute all τ-runs in $O(n/\tau)$ time in a single scan of A. (We assume that T is concatenated with 3τ occurrences of a letter not occurring in T and ignore any τ-run containing such letters.)

We now show how to group all τ-runs by Lyndon roots. By the conditions of the statement, there are no more than $\sigma^{3\tau} \leq n^{3/4}$ distinct strings of length $3\tau - 1$. We generate all of them, and for each of them that is periodic, we compute its Lyndon root in $O(n^{3/4}\log_\sigma n)$ time in total [13]. We store these pairs in a table: the index is the string and the value is the Lyndon root. As the Lyndon root of a τ-run $T[i\,..\,j]$ coincides with the Lyndon root of $T[i\,..\,i + 3\tau - 2]$, we can group all τ-runs as desired using this table in $O(n/\tau)$ time. This is because $T[i\,..\,i + 3\tau - 2]$ can be read in $O(1)$ time in the word RAM model, and so each τ-run is processed in $O(1)$ time.

Similarly, within the same tabulation process, in $O(\tau)$ time, for each distinct string S of length $3\tau - 1$ with Lyndon root R, we compute and store the position i_R of the leftmost occurrence of R in S [13]. Then, for each τ-run $T[a\,..\,b]$ with Lyndon root R and period $p = |R|$, we can obtain, due to periodicity, in $O(1)$ time the starting positions $a + i_R - 1$ and $a + i_R + p - 1$ of the two leftmost occurrences of R in $T[a\,..\,b]$. □

C Proof of Proposition 7.9

Proposition 7.9. *Let $\mathcal{F} \subseteq \Sigma^{\leq \ell}$ and $k \in \mathbb{Z}_{\geq 0}$ with $k = O(1)$. There exists a k-complete family $\mathbf{F}$ for $\mathcal{F}$ such that, for each $F \in \mathcal{F}$:*

- *Every individual set $\mathcal{F}' \in \mathbf{F}$ contains $O(1)$ modified strings with source F.*
- *In total, the sets $\mathcal{F}' \in \mathbf{F}$ contain $O(\min(\ell, \log|\mathcal{F}|)^k)$ modified strings with source F.*

Moreover, if $\mathcal{F}$ consists of substrings of a given length-n text over the alphabet $[0\,..\,n)$, then the family $\mathbf{F}$ can be constructed in $O(n + |\mathcal{F}|)$ space and $O(n + |\mathcal{F}| \min(\ell, \log|\mathcal{F}|)^k)$ time with sets $\mathcal{F}' \in \mathbf{F}$ generated one by one and modified strings within each set $\mathcal{F}' \in \mathbf{F}$ sorted lexicographically.

Our proof follows [89, Section 4] along with the efficient implementation described in [89, Section 5.1]. However, we cannot use these results in a black-box manner since we are solving a slightly more general problem: In [89], the input consists of two strings X and Y, and the output family $\mathbf{F}$ must satisfy the following condition: for every suffix U of X and every suffix V of Y, there exists a set $\mathcal{F}' \in \mathbf{F}$ and modified strings $U^\Delta, V^\nabla \in \mathcal{F}'$ forming a $(U,V)_k$-maxpair. Instead, in Proposition 7.9, we have a set $\mathcal{F}$ of (selected) substrings of a single text. The output family $\mathbf{F}$ must be a k-complete family for $\mathcal{F}$, i.e., satisfy the same condition as before: for every $U, V \in \mathcal{F}$, there exists a set $\mathcal{F}' \in \mathbf{F}$ and modified strings $U^\Delta, V^\nabla \in \mathcal{F}'$ forming a $(U,V)_k$-maxpair. Additionally, we need stronger bounds on the sizes of sets $\mathcal{F}' \in \mathbf{F}$. The original argument [89, Lemma 3] yields $\max_{\mathcal{F}' \in \mathbf{F}} |\mathcal{F}'| = O(|\mathcal{F}|)$ and $\sum_{\mathcal{F}' \in \mathbf{F}} |\mathcal{F}'| = O(|\mathcal{F}| \log^k |\mathcal{F}|)$. Instead, we need to prove that each string $F \in \mathcal{F}$ is the source of $O(1)$ modified strings in any single set $\mathcal{F}' \in \mathbf{F}$ and that there are $O(\min(\ell, \log|\mathcal{F}|)^k)$ modified strings with source F across all sets $\mathcal{F}' \in \mathbf{F}$ provided that $\mathcal{F} \subseteq \Sigma^{\leq \ell}$.

On the high-level, the algorithm behind Proposition 7.9 constructs d-complete families $\mathbf{F}_d$ for $\mathcal{F}$ for each $d \in [0\,..\,k]$. In this setting, the input set $\mathcal{F}$ is interpreted as a 0-complete family $\mathbf{F}_0 = \{\mathcal{F}\}$. For the sake of space efficiency, all families are built in parallel, and the construction algorithm is organised into $2k+1$ *levels*, indexed with 0 through $2k$. Each even level $2d$ is given a stream of sets $\mathcal{F}' \in \mathbf{F}_d$ and its task is to construct the compacted trie $\mathcal{T}(\mathcal{F}')$ for each given $\mathcal{F}' \in \mathbf{F}_d$ (which, in particular, involves sorting the modified strings in $\mathcal{F}'$). Each odd level $2d-1$ is given a stream of tries $\mathcal{T}(\mathcal{F}')$ for $\mathcal{F}' \in \mathbf{F}_{d-1}$, and its task is to output a stream of sets $\mathcal{F}'' \in \mathbf{F}_d$. We finish at an even level to ensure that modified strings within each set $\mathcal{F}' \in \mathbf{F}$ are sorted lexicographically.

The claimed properties of the constructed k-complete family $\mathbf{F}_k$ naturally generalise to analogous properties of the intermediate d-complete families $\mathbf{F}_d$. Moreover, efficient implementation of even levels relies on the following additional invariant: Invariant C.1. *Every set $\mathcal{F}' \in \mathbf{F}_d$ contains a pivot $P^\nabla \in \mathcal{F}'$ such that, for every $F^\Delta \in \mathcal{F}'$, we have $\Delta \subseteq [1\,..\,\mathsf{LCP}(F^\Delta, P^\nabla)] \times \Sigma$.*

The invariant is trivially true for $d = 0$ due to $\Delta = \emptyset$ for every $F^\Delta \in \mathcal{F}'$. Each set will store its pivot.

C.1 Implementation and Analysis of Even Levels

Recall that the goal of an even level $2d$ is to construct $\mathcal{T}(\mathcal{F}')$ for each $\mathcal{F}' \in \mathbf{F}_d$. For this, we rely on Lemma 2.3, which requires sorting the modified strings in $F^\Delta \in \mathcal{F}'$ and computing the longest common prefixes between pairs of consecutive strings.

For the latter task, in the preprocessing, we augment the input text in $O(n)$ time with a data structure for $O(1)$-time LCP queries (Theorem 2.10). Since each modified string $F^\Delta \in \mathcal{F}'$ satisfies $|\Delta| \leq d$, the longest common prefix of any two modified strings in $\mathcal{F}'$ can be computed in $O(1)$ time using up to $2d+1$ LCP queries. In particular, this yields $O(1)$-time lexicographic comparison of any two modified strings in $\mathcal{F}'$, which means that $\mathcal{F}'$ can be sorted in $O(|\mathcal{F}'| \log|\mathcal{F}'|)$ time.

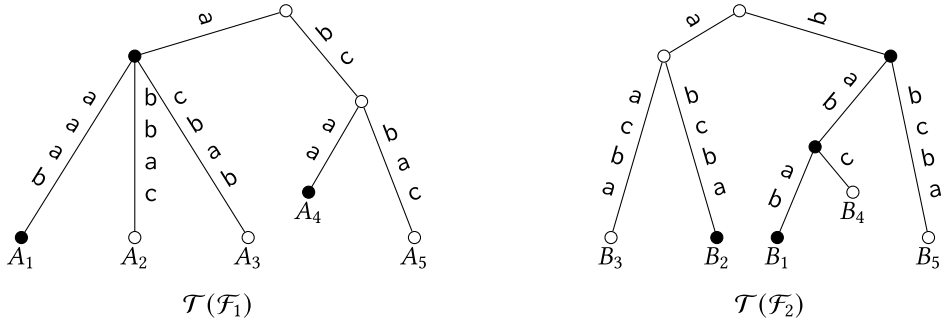


Fig. C1. The compacted tries $\mathcal{T}(\mathcal{F}_1)$ and $\mathcal{T}(\mathcal{F}_2)$ for families $\mathcal{F}_1$ and $\mathcal{F}_2$ from Example 7.7 and Example 7.13, respectively, with heavy (full circles) and light nodes (empty circles) marked. For $i \in \{1, 2\}$, for each light non-leaf node in the trie $\mathcal{T}(\mathcal{F}_i)$, a subfamily of $\mathbf{F}_i$ is created.

Nevertheless, this is too much for our purposes. A more efficient procedure requires buffering the sets $\mathcal{F}' \in \mathbf{F}_d$ into batches $\mathbf{F}'_d \subseteq \mathbf{F}_d$ of total size $\Theta(n + |\mathcal{F}|)$ and using the following result:

Theorem C.2 ([38, Theorem 12]). *Any m substrings of a given length-n text over the alphabet $[0\,..\,n)$ can be sorted lexicographically in $O(n + m)$ time.*

Unfortunately, the possibility of generalising this result to modified substrings remains an open question. A workaround proposed in [89] relies on Invariant C.1. In the lexicographic order of $\mathcal{F}'$, the strings satisfying $F^\Delta \leq P^\nabla$ come before those satisfying $F^\Delta > P^\nabla$. Moreover, in the former group, the values $\mathsf{LCP}(F^\Delta, P^\nabla)$ form a non-decreasing sequence and, in the latter group, these values form a non-increasing sequence. This way, the task of sorting $\mathcal{F}'$ reduces to partitioning $\mathcal{F}'$ into $\mathcal{F}'_p = \{F^\Delta \in \mathcal{F}' : \mathsf{LCP}(F^\Delta, P^\nabla) = p\}$ and sorting each set $\mathcal{F}'_p$. We first compute $\mathsf{LCP}(F^\Delta, P^\nabla)$ for all $F^\Delta \in \mathcal{F}'$ in $O(|\mathcal{F}'|)$ time. Then, in order to construct sets $\mathcal{F}'_p$, we use radix sort for the whole batch $\mathbf{F}'_d \subseteq \mathbf{F}_d$—the keys ($\mathsf{LCP}(F^\Delta, P^\nabla)$ for $F^\Delta \in \mathcal{F}'$) are integers bounded by n. In order to sort the elements of each set $\mathcal{F}'_p$, we exploit the fact that the lexicographic order of a modified string $F^\Delta \in \mathcal{F}'_p$ is determined by $F^\Delta[p + 1\,..\,|F|]$. Moreover, as $p = \mathsf{LCP}(F^\Delta, P^\nabla)$ for the pivot P^∇, we have $\Delta \subseteq [1\,..\,p] \times \Sigma$ by Invariant C.1 and thus $F^\Delta[p + 1\,..\,|F|] = F[p + 1\,..\,|F|]$. Thus, $F^\Delta[p + 1\,..\,|F|]$ is a substring of the input text and hence we can use Theorem C.2 to sort all such substrings arising as we process the batch $\mathbf{F}'_d \subseteq \mathbf{F}_d$, and then classify them back into individual subsets $\mathcal{F}'_p$ using radix sort.

The overall time and space complexity for processing a single batch $\mathbf{F}'_d \subseteq \mathbf{F}_d$ is, therefore, $O(n + |\mathcal{F}|)$. Across all batches, the space remains $O(n + |\mathcal{F}|)$ and the running time becomes $O(n + |\mathcal{F}| \min(\ell, \log |\mathcal{F}|)^d)$ due to $\sum_{\mathcal{F}' \in \mathbf{F}_d} |\mathcal{F}'| = O(|\mathcal{F}| \min(\ell, \log |\mathcal{F}|)^d)$.

C.2 Implementation and Analysis of Odd Levels

Recall that the goal of an odd level $2d - 1$ is to transform a stream of tries $\mathcal{T}(\mathcal{F}')$ for $\mathcal{F}' \in \mathbf{F}_{d-1}$ into a stream of sets $\mathcal{F}'' \in \mathbf{F}_d$. This process is guided by the *heavy-light decomposition* of $\mathcal{T}(\mathcal{F}')$, which classifies the nodes of $\mathcal{T}(\mathcal{F}')$ into *heavy* and *light* so that the root is light and exactly one child of each internal node is heavy: the one whose subtree contains the maximum number of leaves (with ties broken arbitrarily). Each light node w is therefore associated with the *heavy path* which starts at w and repeatedly proceeds to the unique heavy child until reaching a leaf, which we denote $h(w)$. The key property of the heavy-light decomposition is that each node has $O(\log |\mathcal{F}'|)$ light

ancestors. However, since the height of $\mathcal{T}(\mathcal{F}')$ is $O(\ell)$, we can also bound the number of light ancestors by $O(\min(\ell, \log|\mathcal{F}'|))$.

The algorithm constructs the heavy-light decomposition of $\mathcal{T}(\mathcal{F}')$ and, for each light node w, creates a set $\mathcal{F}'_w \in \mathbf{F}_d$ constructed as follows. We traverse the subtree of w and, for each modified string $F^\Delta \in \mathcal{F}'$ with a prefix $\mathsf{val}(w)$, we compute $p = \mathsf{LCP}(F^\Delta, \mathsf{val}(h(w)))$, which is the string-depth of the lowest common ancestor of $h(w)$ and the locus of F^Δ. If $\Delta \subseteq [1\,..\,p] \times \Sigma$, we add F^Δ to $\mathcal{F}'_w$. If additionally $|F^\Delta| > p$, we also add $F^{\Delta \cup \{(p+1, \mathsf{val}(h(w))[p+1])\}}$ to $\mathcal{F}'_w$. This way, we guarantee that $\mathsf{val}(h(w)) \in \mathcal{F}'_w$ forms a pivot of $\mathcal{F}'_w$, as defined in Invariant C.1. Note that each string $F^\Delta \in \mathcal{F}'$ has a prefix $\mathsf{val}(w)$ for $O(\min(\ell, \log|\mathcal{F}'|))$ light nodes w. Hence, each trie $\mathcal{T}(\mathcal{F}')$ for $\mathcal{F}' \in \mathbf{F}_{d-1}$ is processed in $O(|\mathcal{F}'|\min(\ell, \log|\mathcal{F}'|)) = O(|\mathcal{F}'|\min(\ell, \log|\mathcal{F}|))$ time and $O(|\mathcal{F}'|) = O(|\mathcal{F}|)$ space. Across all $\mathcal{F}' \in \mathbf{F}_{d-1}$, this yields $O(|\mathcal{F}|\min(\ell, \log|\mathcal{F}|)^d)$ time and $O(|\mathcal{F}|)$ space.

It remains to prove that $\mathbf{F}_d$ satisfies all the conditions of Proposition 7.9. First, we argue about the bullet points. Each modified string $F^\Delta \in \mathcal{F}'$ gives rise to at most two modified strings added to a single set $\mathcal{F}'_w$ and $O(\min(\ell, \log|\mathcal{F}|))$ modified strings across sets $\mathcal{F}'_w$ for light nodes w in $\mathcal{T}(\mathcal{F}')$. Since each string $F \in \mathcal{F}$ is the source of $O(1)$ modified strings in any single set $\mathcal{F}' \in \mathbf{F}_{d-1}$ and $O(\min(\ell, \log|\mathcal{F}|)^{d-1})$ modified strings across all sets $\mathcal{F}' \in \mathbf{F}_{d-1}$, it is the source of $O(1)$ modified strings in any single set $\mathcal{F}'_w \in \mathbf{F}_d$ and $O(\min(\ell, \log|\mathcal{F}|)^d)$ modified strings across all sets $\mathcal{F}'_w \in \mathbf{F}_d$.

Finally, we shall argue that $\mathbf{F}_d$ is indeed a d-complete family. The modified strings in $\mathcal{F}' \in \mathbf{F}_{d-1}$ have sources in $\mathcal{F}$ and up to $d-1$ modifications. Each string inserted to $\mathcal{F}'_w \in \mathbf{F}_d$ retains its source and may have up to one more modification, for a total of up to d. Next, consider two strings $U, V \in \mathcal{F}$. Since $\mathbf{F}_{d-1}$ is a $(d-1)$-complete family, there is a set $\mathcal{F}' \in \mathbf{F}_{d-1}$ and modified strings $U^\Delta, V^\nabla \in \mathcal{F}'$ forming a $(U,V)_{d-1}$-maxpair. Let v be the node of $\mathcal{T}(\mathcal{F}')$ representing the longest common prefix of U^Δ and V^∇, and let w be the lowest light ancestor of v (so that v lies on the path from w to $h(w)$). By Fact 7.5, we have $p := \mathsf{LCP}(U^\Delta, V^\nabla) = \mathsf{LCP}_{d-1}(U,V)$. Definition 7.3 further yields $\Delta, \nabla \subseteq [1\,..\,p] \times \Sigma$. If $\mathsf{LCP}_d(U,V) = p$, then $U^\Delta, V^\nabla \in \mathcal{F}'_w$ form a $(U,V)_d$-maxpair. Otherwise, let $c = \mathsf{val}(h(w))[p+1]$ and note that $U[p+1] \neq V[p+1]$. If $U[p+1] \neq c \neq V[p+1]$, then $U^{\Delta \cup \{(p+1,c)\}}, V^{\nabla \cup \{(p+1,c)\}} \in \mathcal{F}'_w$ form a $(U,V)_d$-maxpair. If $U[p+1] \neq c = V[p+1]$, then $U^{\Delta \cup \{(p+1,c)\}}, V^\nabla \in \mathcal{F}'_w$ form a $(U,V)_d$-maxpair. Symmetrically, if $U[p+1] = c \neq V[p+1]$, then $U^\Delta, V^{\nabla \cup \{(p+1,c)\}} \in \mathcal{F}'_w$ form a $(U,V)_d$-maxpair. Thus, in all four cases, $\mathcal{F}'_w \in \mathbf{F}_d$ contains a $(U,V)_d$-maxpair. This completes the proof of Proposition 7.9.

Received 12 September 2024; revised 7 October 2025; accepted 28 October 2025